Introduction
Welcome

Welcome to the developer support package for the Atari
Falcon030. This kit contains a machine with 16 Megabytes of
RAM (14 Megabytes are used), a harddisk, software on the
harddisk and on floppies, and the documentation package
that you are reading.

The DSP folder on the harddisk contains the files
ASM56000.TTP, DSPLNK.TTP, CLDLOD.TTP, README and
DSPBIND.H which is the binding needed to use the DSP
Support Routines described in this manual.

Note: Some compiler systems require different binding
structures than used in DSPBIND.H

Overview

From the point of view of an application writer, the most important thing to realize about the Atari Falcon030 is that it is an ST compatible TOS machine. This means that software written to take advantage of the operating system features via operating system calls of the previous machines will work without modification. In fact, a great deal of effort was expended to insure that a large percentage of software that runs on previous machines will continue to operate.

The hardware changes that the user will see between the Atari Falcon030 and previous TOS machines involve improved video and sound abilities.

The video system has the following characteristics:

1) The system supports both VGA and broadcast system monitor types. For this document, "broadcast type monitor" means a TV as well as an analog RGB monitor such as the SC1224.

2) The number of vertical lines can be either 200 or 400 (240 or 480 on VGA). This is done by using interlace on broadcast monitors to get 400 lines and doubling each line on VGA monitors to get 240.

3) The number of horizontal pixels can be approximately either 640 or 320.

4) The number of bit planes can be either one, two, four or eight.

5) Characteristics numbered 1-4 can be mixed in any combination. (except 320 wide 1-bit plane)

6) The color palette is 262144 in 1, 4 or 8 bit per pixel mode and 4096 in 2 bit per pixel mode.

7) Overscan is available in all broadcast video modes. Overscan will multiply the pixel count by 1.2

8) A 16 bit per pixel, true color mode exists that will operate in all resolutions except 640 pixel wide VGA mode. All of these modes can be accessed via the GEM VDI. In the case of the true color mode there is no color palette to allow for changing the color of pixels that

---

have already been drawn. The GEM VDI provides 256 virtual pens to use for drawing. These pens act just like the physical pens in the other modes except that once a pixel is drawn, it cannot be changed using vs_color().

The sound system has the following components:

1) 56001 Digital Signal Processor
2) DMA sound engine that can playback or record one, two, three or four 16 bit stereo channels at 12.5,25 or 50 kHz.
3) 16 bit stereo codec allowing both input and output of sound via built-in headphone and microphone jacks.
4) An external port (DSP) that allows external I/O for a wide variety of purposes. The details of how these various components can be used and in what combinations are given in other documents.

Atari MultiTOS
User Interface Guidelines
Application Elements

User-friendly GEM applications should provide the user with a consistent, predictable means of interacting with the computer. The most popular applications to-date have always been those that the user feels at home with, because of general familiarity with other applications that they have previously used. User interface design is a critical consideration during product development and should be well thought out before actually sitting down and laying out and coding the interface.

The basic elements of a GEM application are the menu bar, the application's window (or windows), dialog boxes, alert boxes, and if the application warrants them, toolbox windows. GEM applications may optionally install their own desktop background, which is swapped out by the AES to reflect the foreground application.

The Menu Bar

Applications should normally consist of a MENU BAR, which will generally have the titles from left to right, "Prgname", "File", "Edit", and then the additional application-specific main menu titles. "Prgname" should be replaced with the application name so that users can quickly identify which application's menu bar they are looking at.

For user convenience, the standard entries under "File" should start with "New", "Open...", followed by other start-oriented operations, then in the next section of the menu, "Close", "Save", "Save as...", and the other application-specific end-oriented functions. The next section down should be used for other file operations such as "Import..." and "Export...". This should be followed by the menu items for printing, usually "Page Setup...", then "Print...". The last item under "File" should always be "Quit".

Note -- A menu item must be followed by an
ellipsis to indicate that additional action or input
will be required by the user to carry out the
requested task. For instance, "Save" indicates that
the file will be saved directly, using the current
name, whereas "Save as..." will require the
additional input of a filename.

The "Edit" menu should start with "Undo", then in the next
section, "Cut", "Copy", "Paste", and "Delete". The rest of the
"Edit" menu is usually application-specific, but the next
menu item, if used should be "Select all".

If applicable, the fourth main menu title should be "Options",
where menu items such as "Document defaults...", or
"Preferences..." should appear.

Note -- Menu titles and items should never be
displayed in all uppercase letters. Menu titles
should have one space before and after each title.
There should be one space to the left of menu
items.

Keyboard Equivalents for Menu Items

The standard sytem-wide keyboard equivalents that should
be used system-wide for no other purpose other than those
listed are:

[Control-N]   New
[Control-O]   Open
[Control-W]   Close
[Control-S]   Save
[Control-P]   Print
[Control-Q]   Quit

[Control-X]   Cut
[Control-C]   Copy
[Control-V]   Paste
[Control-A]   Select all

[Control-F]   Find
[Control-H]   Replace
[Control-G]   Find next

[Delete]   Delete
[Undo]   Undo
[Help]   Invoke help

The [Alternate] key is used as a character modifier on many keyboards to access the necessary extended characters in applicable countries, and should not be used for keyboard equivalents in most cases.

Windows

The primary stage for user interaction with the application is the window. Most of the user input, whether typing, drawing, or editing, is performed in the confines of windows. All of an application's output should be constrained to the application's own windows only. See the VDI and AES manuals for further information regarding window work areas and clipping rectangles.

Document windows should have at a minimum, a mover/title bar so that even if the window is not resizable, the user can move the window off to the side of the desktop to have access to other items. The other window elements are the Info bar, Closer, Sizer, Full box, Sliders, and Arrows. The general use of these is apparent in the GEM Desktop. It should be noted that GEM sliders are always proportional so that the user has constant feedback as to the percentage of the document that is being viewed.

Operating system calls allow every element of windows to be set to any color and fill pattern. The user generally selects these attributes using the Window Colors CPX in the Control Panel and they should not be altered by an application. The first 16 color entries should be reserved for use by the system for drawing elements for which the user has set preferences.

Keyboard Equivalents for Cursor Movement Inside Windows

The system-wide standard for keyboard cursor manipulation is as follows:

[Control-Left/Right Arrow]   Move cursor to beginning of word to the left/right

[Control-Backspace]   Delete from cursor position to start of next word to the left

[Control-Delete]   Delete from cursor position to start of next word to the right

[Control-ClrHome]   Move cursor to beginning of document

[Shift-ClrHome]   Move cursor to end of document

[Shift-Delete]   Delete line

Dialog Boxes

Dialog boxes are used for modal input, that is, input that the user must provide before any further processing may be done. They are generally used for parameter setting and other selections that require the undivided attention of the user. They should never be used for on-going information or status output, as it would interfere with the normal real-time user interaction with the system.

Alerts

Alerts should be used to call the user's attention to conditions that develop that require immediate user knowledge. The simplest and most common would be an alert notifying the user that he is quitting an application without having saved an open, modified document. Alerts should also be used to notify the user that a time-consuming or unalterable function is about to be performed.

Alerts usually have two or three buttons that allow the user to make some sort of decision based on the information provided. Alerts with only one button are very frustrating to the user, as it implies a lack of control over what is about to

happen. The general rule for alerts is to have the "OK" button to the left of the "Cancel" button. "Cancel" should always be capitalized, and "OK" is uppercase.

> **Note -- Buttons in general should be capitalized words, not all uppercase.**

Toolbox Windows

Toolbox Windows are a special class of window that are used for providing the user with non-modal control or information. The most common use would be for drawing tool selection in a paint program, or color selection. The tools are usually shown as logical groups of icons that the user can easily associate with their functions. Another use of this type of window is continual status output, such as the progress of a file download or recalculation time.

Other General Notes

Applications should make no assumptions on what type of system the user will have. Be able to deal with any screen size and color resolution. Use the operating system calls to determine the screen dimensions and system capabilities to provide the user with the richest computing experience possible. Users have grown to expect unsurpassed ease of use from applications available for Atari computers. If you have any questions regarding user interface design for Atari computers, please feel free to call your developer support representative.

Game/Entertainment Software Guidelines

The following points should be followed...

• Installable on a harddisk

• Should be able to be launched from any video resolution

• The user should be presented with a single executable file; leave ancillary data files, high score files, etc. inside a companion folder.

• Allow the user to exit and return to the desktop exactly where and how s/he left off.

• Use the enhanced joystick for all joystick-oriented games; CX-40 style controls should not be supported.

• Ideally, where possible, allow the game to be run in a window; this is well-suited for users that want to play games in the MultiTOS multi-tasking environment (such as while downloading a file).

• We expect most users to run in 640x480x256 color mode; you may want to keep this in mind.

• If you use the O/S call, vr_trn_fm() (transform form), you can easily convert video data from standard form to the correct form for the current resolution.

Menu Library Enhancements

Introduction

This section describes the additional features of the Menu Library. All enhancements are backwards compatible with previous versions of the AES, so existing applications will continue to work. The new features will work on all machines with an AES version number of 3.3 and up.

The enhancements to the Menu Library are:
- Heirarchical menus are now supported.
- Pop-Up Menus are now supported.
- Scrolling menus are supported for pop-up menus and submenus. Scrolling for the first level menus of a menu bar are not supported.

Heirarchical Menus

Heirarchical menus allow a menu item to be the title of a submenu. Menu items with a right arrow signify that a submenu is attached. Heirarchical menu items must be of the type G_STRING. As a rule, the Desk Menu of a menu bar is not allowed to have submenus.

Two delay values are used to prevent the rapid appearance and disappearance of submenus:
- Submenu Display Delay
  This delay is used to prevent the rapid flashing of submenus as the mouse pointer is dragged thru a menu item with an attached submenu. The mouse pointer must remain within the menu item for the delay period before the submenu is displayed. The default Submenu Display Delay is 1/5 of a second. menu_settings can be used to inquire the current delay value, or to set a new delay.
- Submenu Drag Delay
  This delay is used to prevent the disappearance of the submenu as the mouse pointer is dragged toward the submenu from a menu item. The default Submenu Drag Delay is 10 seconds. menu_settings can be used to inquire the current delay value, or to set a new delay.

There are several actions that will cancel the Submenu Drag Delay prematurely:
1) If the mouse pointer is dragged away from the direction of the submenu, the submenu will disappear.
2) If the mouse pointer remains in the same position after the drag has begun, the submenu will also disappear.
3) If the user clicks on the left mouse button before the mouse pointer has entered the submenu, the system will return to the application the menu item that started the drag.
4) If the mouse pointer is dragged vertically into another menu item, the submenu will disappear.

As a rule, only one level of heirarchical menus should be used. The actual number of recursions possible is currently set to 4.

Pop-Up Menus

Pop-up menus are menus that are not in the menu bar. They can be placed anywhere on the screen and once displayed, act like any other menu.

Scrolling Menus

When the number of menu items exceeds the menu scroll height, a scroll indicator appears at the bottom of the menu. The scroll indicators are displayed as UP or DOWN ARROWS. Clicking on the bottom arrow will scroll the menu items. When the last item is shown, the DOWN ARROW indicator disappears. Note that as soon as the menu started scrolling, the UP ARROW indicator appeared at the top of the menu. This is to show that there are now menu items in that direction. The default menu scroll height is 16. menu_settings can be used to inquire the current menu scroll height, or to set a new menu scroll height.

When the user clicks and holds down the left mouse button, there is a 1/4 of a second delay after one menu item has scrolled. After the delay, scrolling continues uninterrupted. This delay is used to prevent rapid scrolling for those just clicking on the scroll indicators. menu_settings can be used

to inquire the current delay, or to set a new delay.

Another delay value is used to slow down the scrolling speed. This prevents the menu items from scrolling by too fast. menu_settings can be used to inquire the current delay, or to set a new delay.

Pop-up menus and submenus might consist of objects other than G_STRINGS. Such a menu might consist of user-defined objects that display the system's fill patterns. The system cannot scroll non-G_STRING object types. Scrolling non-G_STRING object types will crash the system. Pop-up menus and submenus containing non-G_STRING object types should have its scroll_flag field set to FALSE.

The first-level menus of a menu bar are set to be non-scrollable. This is due to the parent-child relationships between the menu titles, menus and menu items. Therefore, scrolling is applicable only to pop-up menus and submenus.

Using the Extended Menu Library

The existing Menu Library functions are still applicable to pop-up menus and submenus. The Menu Library will continue to have the following responsibilities:
- displaying the appropriate menu bar for each active application
- enabling and disabling menu items
- displaying check marks in menus
- returning a highlighted menu title to its normal state
- displaying context-sensitive menu text
- displaying a desk accessory's name on the Desk Menu

To use pop-up menus and submenus in one's application, create an object tree consisting of a G_BOX and as many G_STRINGS within the G_BOX as required. The G_BOX is the menu and the G_STRINGS are the menu items. An object tree is not limited to just one menu and can contain one, two or more menus. If a menu item is expected to have a submenu attachment, the G_STRING must be padded with blanks to the width of the menu.

The object tree does not need to be created with the Resource Construction Set. It can be created during runtime by the application. However, the programmer is responsible for this procedure.

Attaching a submenu to a menu item is done by calling menu_attach. A submenu is associated to a menu item by placing a right arrow two characters in from the right edge. Any characters at that location will be overwritten.

The high-byte of the object's type field is used to store an internal Menu ID. The values between 128 and 192 are used by the new menu sytem.

In addition, Bit 11 of the object's ObFlag field will be set. Bit 11 is defined as: '#define SUBMENU 0x800'. Applications using the Extended Object Type AND SUBMENUS should first check the object's ObFlag field to see if the value in the Extended Object Type is a submenu attachment.

Each process can have up to 64 unique submenu attachments. Attaching the same submenu to multiple menu items counts as one attachment.

In addition to attaching a submenu, menu_attach can be used to change or remove a submenu. menu_attach can also be used to find out what submenu, if any, is attached to a menu item. menu_istart can be used to set and get the starting menu item of a submenu.

menu_settings can be used to set the menu delay values and to set the height at which pop-up menus and submenus will start to scroll.

Using a Menu Bar

When the user chooses an item, the Screen Manager writes a message to the pipe. Control then returns to the application, which must read the pipe.

The pipe message contains the following:
- a code indicating that it is a menu message (MN_SELECTED)
- the object index of the menu title selected
- the object index of the menu item chosen
- the object tree of the menu item chosen (NEW)
- the object index of the parent of the menu item (NEW)

(If the user does not choose an item, or if the user selects a disabled menu item, the Screen Manager does not write a message to the pipe.)

After processing the chosen item, the application makes a Menu Library call to dehighlight the menu title and waits for the next message to come through the message pipe.

Extended Menu Library Routines

The additions to the Menu Library routines are:
- menu_popup: Displays a menu anywhere on the screen. Clipping is performed for a standard menu. Menus with user-defined objects will have to perform their own clipping.
- menu_attach: Lets an application attach, change, remove or inquire about a submenu associated with a menu item.
- menu_istart: Lets an application set and inquire the starting menu item of a pop-up menu or submenu
- menu_settings: Lets an application set and inquire the delay and height parameters of the submenus.

---

menu_popup and menu_attach use a new structure for
passing and receiving submenu data. The MENU structure is
defined as follows:

```
typedef struct _menu
{
    OBJECT *mn_tree;     - the object tree of the menu
    WORD   mn_menu;      - the parent object of the menu items
    WORD   mn_item;      - the starting menu item
    WORD   mn_scroll;    - the scroll field status of the menu
```

0 - The menu will not scroll

!0 - The menu will scroll if the
number of menu items exceed
the menu scroll height. The
non-zero value is the object at
which scrolling will begin. This
will allow one to have a menu in
which the scrollable region is
only a part of the whole menu.
The value must be a menu item
in the menu.

menu_settings can be used to
change the menu scroll height.

NOTE: If the scroll field status is !0, the
menu items must consist
entirely of G_STRINGS.

```
    WORD   mn_keystate;  - The CTRL, ALT, SHIFT Key state
```
at the time the mouse button was
pressed.

```
}MENU;
```

menu_settings uses a new structure for setting and inquiring
the submenu delay values and the menu scroll height. The
delay values are measured in milliseconds and the height is
based upon the number of menu items.

```
typedef struct _mn_set
{
    LONG Display;   - the submenu display delay
    LONG Drag;      - the submenu drag delay
```

```
        LONG  Delay;    - the single-click scroll delay
        LONG  Speed;    - the continuous scroll delay
        WORD  Height;  - the menu scroll height
}MN_SET;
```

- Submenu Display Delay:
  The delay is used to prevent the rapid flashing of submenus as the mouse pointer is dragged thru a menu item with an attach submenu. The default value is 200 milliseconds (1/5th of a second).

- Submenu Drag Delay:
  The delay is used to prevent the disappearance of the submenu as the mouse pointer is dragged toward the submenu from a menu item. The default value is 10000 milliseconds (10 seconds).

- Single-Click Scroll Delay:
  This is the delay period after one menu item has initiallly scrolled. After the delay, scrolling continues at the rate specified by the Continuous Scroll Delay. The delay is used to prevent rapid scrolling for those just clicking on the scroll indicators. The default value is 250 milliseconds (1/4th of a second).

- Continuous Scroll Delay:
  This is the delay period after each menu item has scrolled. The delay is used to slow down the scrolling speed. The default value is 0 milliseconds.

- Menu Scroll Height:
  This value is the height at which a pop-up menu or a submenu will start to scroll if its scroll field is TRUE. The default value is 16 menu items.

The following text describe these routines.

MENU_POPUP

Allows an application to display a popup menu anywhere on
the screen. The popup menu may also have submenus. If
the number of menu items exceed the menu scroll height, the
menu may also be set to scroll. menu_settings can be used to
set the height at which all menus will start to scroll.

Parameters:
```
control(0)  =  36
control(1)  =  2
control(2)  =  1
control(3)  =  2
control(4)  =  0
int_in(0)   =  me_xpos
int_in(1)   =  me_ypos
int_out(0)  =  me_return
addr_in(0)  =  me_menu
addr_in(1)  =  me_mdata
```

- me_xpos - the left edge of where the starting menu item
  will be displayed
- me_ypos - the top edge of where the starting menu item
  will be displayed
- me_return - a coded return message
  0 - FAILURE: The data returned by me_mdata is invalid
  1 - SUCCESS: The data returned by me_data is valid
  FAILURE is returned if the user did not click on an
  enabled menu item.
- me_menu - pointer to the pop-up MENU structure. The
  structure must be initialized with the object tree of the
  pop-up menu, the menu object, the starting menu item
  and the scroll field status.
- me_mdata - pointer to the data MENU structure. If
  menu_popup returns TRUE, me_mdata will contain
  information about the submenu that the user selected.
  This includes the object tree of the submenu, the menu
  object, the menu item selected and the scroll field status
  for this submenu.

Sample call to C language binding:
```
me_return = menu_popup(MENU *me_menu, word me_xpos,
                       word me_ypos, MENU *me_mdata);
```

MENU_ATTACH

Allows an application to attach, change, remove or inquire about a submenu associated with a menu item.

Parameters:
```
control(0) = 37
control(1) = 2
control(2) = 1
control(3) = 2
control(4) = 0

int_in(0)  = me_flag
int_in(1)  = me_item

int_out(0) = me_return

addr_in(0) = me_tree
addr_in(1) = me_mdata
```

- me_flag - the action to be performed by menu_attach.

The options for me_flag are:

0   Inquire data about the submenu that is associated with the menu item. The data concerning the submenu is returned in me_mdata.

1   Attach or change a submenu associated with a menu item. me_mdata must be initialized by the application. The data must consist of the object tree of the submenu, the menu object, the starting menu item and the scroll field status. Attaching a NULLPTR structure will remove the submenu associated with the menu item. There can be a maximum of 64 associations per process. Bit 11 of the object's ObFlag will be set if a submenu is actually attached.

2   Remove a submenu associated with a menu item. me_mdata should be set to NULLPTR. Bit 11 of the object's ObFlag will be cleared.

- me_item - the menu item that the submenu will be attached to
- me_return - a coded return message
  0 - FAILURE: the submenu was not attached for whatever reasons
  1 - SUCCESS: the submenu was attached, changed or removed successfully
- me_tree - the object tree of the menu item that will have a submenu attach to
- me_mdata - pointer to the MENU structure. The contents of me_mdata are dependant upon the value of me_flag:
  0 Upon return from menu_attach, me_mdata will contain the MENU data regarding the submenu associated with the menu item.
  1 me_mdata must be initialized with the new submenu MENU data. The submenu will be attached to the menu item - me_item.
  2 me_mdata should be set to NULLPTR. The submenu associated with the menu item will be removed.

Sample call to C language binding:
```
me_return = menu_attach(word me_flag, object *me_tree,
                        word me_item, MENU *me_mdata );
```

MENU_ISTART

Allows an application to set or inquire the starting menu item of a submenu that is associated with a menu item. The submenu is shifted vertically so that the starting menu item is aligned with the menu item that is associated with this submenu.

Parameters:
    control(0) = 38
    control(1) = 3
    control(2) = 1
    control(3) = 1
    control(4) = 0

    int_in(0) = me_flag

```
int_in(1)  = me_imenu
int_in(2)  = me_item

int_out(0) = me_return

addr_in(0) = me_tree
```

- me_flag - the action to be performed by menu_istart
  0 Inquire the starting menu item for the submenu
  1 Set the starting menu item for the submenu to be me_item
- me_imenu - the menu object of the submenu that is either to be set or inquired
- me_item - the starting menu item that is either to be set or inquired
- me_return - a coded return message
  0 - FAILURE: the submenu is not associated with a menu item. The submenu must be attached via menu_attach before this call can be made.
  >0 - SUCCESS: the starting menu item is currently set to this value.
- me_tree - the object tree of the menu item that we are setting or inquiring about

Sample call to C language binding:

```
me_return = menu_istart(word me_flag, object *me_tree,
                        word me_imenu, word me_item );
```

MENU_SETTINGS

Allows an application to set or inquire the submenu delay values and the menu scroll height value.

Parameters:
```
control(0)  = 39
control(1)  = 1
control(2)  = 1
control(3)  = 1
control(4)  = 0

int_in(0)   = me_flag

int_out(0)  = me_return
```

addr_in(0) = me_values

- me_flag - the action to be taken by menu_settings
  0 Inquire the current delay and menu scroll height values.
  1 Set the delay and menu scroll height values
- me_return - always returns 1 (one)
- me_values - pointer to the MN_SET structure.
  me_values is dependant upon the value of me_flag:
  0 Upon the return of menu_settings, me_values will contain the current delay and menu scroll height values.
  1 me_values must be initialized. The delay and menu scroll height values will be set to those values found in me_values. A value set to NIL will be ignored.

Sample call to C language binding:
```
me_return = menu_settings(word me_flag,
                          MN_SET *me_values );
```

AES Supplemental Documentation

The following text contains documentation supplemental to the existing AES manual, and clarifications of existing documentation related to heirarchical submenus and the menubar.

Supplement to: MN_SELECTED

GEM AES uses this message to notify an application that a user has selected a menu item.

- word 0   = 10
- word 3   = the object index of the menu title selected
- word 4   = the object index of the menu item selected
- word 5,6 = the object tree of the menu item selected
- word 7   = the parent object of the menu item selected

Pop-Up Menus

- The button on a dialog box that brings up a pop-up menu should be shadowed.

- It would be nice if the pop-up menu was shadowed also.
- While the pop-up menu is displayed, if it has a title, the title should be inverted.
- The pop-up menu should be aligned on a byte boundary. This speeds up the drawing of the menu considerably.
- The pop-up menu will be shifted vertically in order to line up the start object with the given coordinates.
- If the menu exceeds the top of the screen, it will be shifted down.
- No horizontal adjustments will be done to the menu.

Submenus

- Menu items expecting a submenu attachment must be of type G_STRING.
- Menu items should be padded with blanks to the width of the menu.
- Menu items expecting a submenu attachment should not have any keyboard short-cut characters.
- Submenus will automatically be displayed on a byte boundary.
- The menu will be shifted vertically to align the start object with the menu item. In addition, the menu will be shifted to remain entirely on the screen in the vertical plane.
- The submenu will be displayed at the right edge of the menu item. If the menu extends off the edge of the screen, the menu will be displayed to the left of the menu item. If it exceeds the left edge, the menu will be shifted right a character at a time, until it fits.
- There can be a maximum of 64 submenu attachments per process.
- A menu item with an attached submenu uses the high-byte of its object type field. Values 128 thru 192 are used by the submenu menu system.
- A menu item with an attached submenu will have Bit 11 of its object flag field SET.
  The bit is defined as: #define SUBMENU 0x800

- Applications using the Extended Object Type should check the object's ObFlag field to see if Bit 11 is SET. If the bit is SET, the menu item has a submenu attached.
- A submenu should not be attached to itself.
- Attaching a submenu to different menu items counts as one attachment. There will only be one scroll flag and one start object.
- As a user interface guideline, there should only be one level of heirarchical menus. The system currently allows up to four levels of recursion.
- menu_istart works only on submenus attached with menu_attach.
- Menu items with attached submenus cannot have keyboard shortcuts.

Scrolling Menus

- In order to scroll properly, all menu items must be G_STRINGS. Menus that contain objects other than G_STRINGS should set the scroll flag to 0.
- The first-level menus of a menu bar are not scrollable.
- Pop-up menus and submenus with greater than sixteen items can be scrolled if their scroll flag is set. The number of items to scroll at can be adjusted with menu_settings.
- If the pop-up menu or submenu is designed to be a toolbox, (ie: fill patterns), set the scroll flag to FALSE.
- Setting the scroll flag to one of the menu items will initiate scrolling from that menu item if the number of items exceeds the menu height limit.
- One should NOT set the scroll object to the last menu item of a menu.
- Setting the scroll object to a value less than the first menu item defaults to the first menu item.
- Setting the scroll object to a value greater than or equal to the last menu item defaults to the first menu item.

Structure for Passing Menu Data

```
typedef struct _menu
{
    OBJECT  *mn_tree;        /* Object tree of the menu      */
    WORD    mn_menu;         /* Parent of the menu items     */
    WORD    mn_item;         /* Starting menu item           */
    WORD    mn_scroll;       /* scroll flag for the menu     */
    WORD    mn_keystate;     /* CTRL, ALT, SHIFT Keystate*/
}MENU;
```

Structure for the Menu Settings

```
typedef struct _mn_set
{
    LONG    Display;         /* The display delay       */
    LONG    Drag;            /* The drag delay          */
    LONG    Delay;           /* The Arrow Delay         */
    LONG    Speed;           /* The scroll speed delay  */
    WORD    Height;          /* The menu scroll height */
}MN_SET;
```

```
WORD  menu_popup(MENU *Menu, WORD xpos,
                    WORD ypos, MENU *MData);
WORD  menu_attach(WORD flag, OBJECT *tree,
                    WORD item, MENU *Menu);
WORD  menu_istart(WORD flag, OBJECT *tree,
                    WORD menu, WORD item);
WORD  menu_settings(WORD flag, MN_SET *Values);
```


Atari Falcon030
Hardware Reference Guide
Version: 2.1
Date: October 1, 1992

Introduction

The Atari Falcon030 is a new generation of Atari TOS-compatible computers. It is based around a Motorola 68030 32 bit microprocessor and includes an optional Motorola 68881/2 Floating point coprocessor, a 16MHz - 16 bit BLiTTER, and a 32 MHz Motorola 56001 Digital Signal Processor.

The Atari Falcon030 hardware specification can be summarized as follows:

CPU: 68030, 16MHz

FPU: Socket for optional 68881 or 68882 running at 16 MHz.

RAM: Custom module. 1 to 16 MBytes of RAM.

ROM: 512 KBytes.

BLiTTER: Graphics coprocessor running at 16MHz.

Video:

	Non-Overscan	Overscan
Horizontal	320	384
	640	768
Vertical	200	240
	400	480

	Bit Planes	Colors	Palette
ST Low-res	4	16	4,096
ST Med-res	2	4	4,096
ST High-res	1	2	4,096
Atari Falcon030	8	256	262,144
	4	16	262,144
	1	2	262,144
	16	65536	N/A

All modes can also be Genlocked, to provide multi-media capabilities on monitors or Televisions. The true color modes also directly support overlays.

An on-board RF modulator allows for direct connection to TVs. Monitor connector allows connection to VGA monitors, ST monochrome, or color monitors (via an adaptor plug).

Horizontal scrolling is supported, compatible with STE.

Sound:
Built in stereo 16-bit Analog to Digital Convertor (ADC).
Built in stereo 16-bit Digital to Analog Convertor (DAC).
Stereo microphone input and stereo headphone output
    jacks. Internal speaker (mono).
3 Channel PSG sound (compatible with ST).
8 Channel 16 bit PCM digital record/playback I/0.
Stereo 8 bit PCM sound (compatible with TT030, STE, and
    MSTE).
Digital Audio/DSP connector.
Sophisticated multiplexer connects DSP, Codec, DMA, and
external I/O connector.

DSP: 32MHz Motorola 56K Digital Signal Processor with
32Kx24 zero wait-state SRAM.

I/O:
Parallel port.
Modem/RS232 port.
MIDI in.
MIDI out.
Cartridge port.
SCSI II (50 pin connector) with DMA.
LAN Local area network (compatible with TT030
    and MegaSTE).

Joysticks: Two STE compatible enhanced joystick ports
supporting four paddles, a light gun, and up to 21 buttons
each. (See keypad documentation)

FDD: 1.44 Mbyte Floppy Disk Drive.

HDD: Internal optional hard disk drive on IDE bus.

Keyboard: 94/95 key keyboard

Mouse: 100 DPI mouse supplied as standard.

Other:
Real time clock with battery backed, non-volatile RAM.
Optional internal HDD.
Internal expansion connector.

Mechanical Specification

```
Connectors

Type       Pins   Type    #   Description
-------------------------------------------------------------
Rear panel:
DIN 5        5     Female  1   MIDI in
DIN 5        5     Female  1   MIDI out
DB25        25     Female  1   Parallel port
DB9          9     Male    1   Modem / Serial port
SCSI II     50     Female  1   SCSI II
DB19        19     Male    1   Video out / Genlock
Mini-Jack    3     Female  1   Stereo Headphone out
Mini-Jack    3     Female  1   Stereo Microphone in
DB26        26     Female  1   DSP/Digital Audio interface
RCA          2     Female  1   RF Modulator
MiniDIN      9     Female  1   LAN
Reset switch

Left Side panel:
Custom      40             1   Cartridge port
DB15        15     Male    2   STE compatible enhanced joysticks

Underside:
DB9          9     Male    2   ST compatible joystick/mouse ports

Internal:
Headers   30+50    Male    1   DRAM expansion board
Headers   30+50    Male    1   Internal bus expansion
Header      44     Male    1   Internal IDE connection
Header      34     Cable   1   Internal Floppy Disk Drive

Other:
Rechargeable cell on motherboard for battery backed RAM/RTC
      Lasts over 10 years
Internal speaker
```

Internal Expansion Port

The Atari Falcon030 has a full featured, internal expansion bus.

J20. 30 pin, dual row, upright male header

Pin#	Signal	Pin#	Signal
1	D14	2	D13
3	D12	4	D11
5	D10	6	D9
7	D8	8	D7
9	D6	10	D5
11	D4	12	D3
13	D2	14	D1
15	D0	16	D15
17	GND	18	GND
19	GND	20	CPUBGO
21	EINT1	22	CPUBGI
23	500KHZ	24	n/c
25	MFP_IEI	26	MFP_INT
27	EINT3	28	VCC
29	VCC	30	VCC

J19. 50 pin, dual row, upright male header

Pin#	Signal	Pin#	Signal
1	GND	2	GND
3	BGK	4	AS
5	LDS	6	UDS
7	RXW	8	DTACK
9	FC2	10	FC1
11	FC0	12	BMODE
13	n/c	14	IACK
15	BG	16	BR
17	RESET	18	HALT
19	BERR	20	IPL0
21	IPL1	22	IPL2
23	CPUCLK	24	VCC
25	VCC	26	A23
27	A22	28	A21
29	A20	30	A19
31	A18	32	A17
33	A16	34	A15
35	A14	36	A13
37	A12	38	A11
39	A10	40	A9
41	A8	42	A7
43	A6	44	A5
45	A4	46	A3
47	A2	48	A1
49	EXPAND	50	n/c

10.92

1.6 THK (PCB)

2 (MAX)

5.5 (MAX)

2.54

110

52

44.75

2.54

2.35

2.26

44.53

50 PIN SOCKET CONN
(SUGGSTED VENDOR AND
P/N JAE PS-505D-DATS1)

PIN #1

30 PIN SOCKET CONN
(SUGGESTED VENDOR AND
P/N JAE PS-3050-DATS)

PIN #1

All dimensions in millimeters
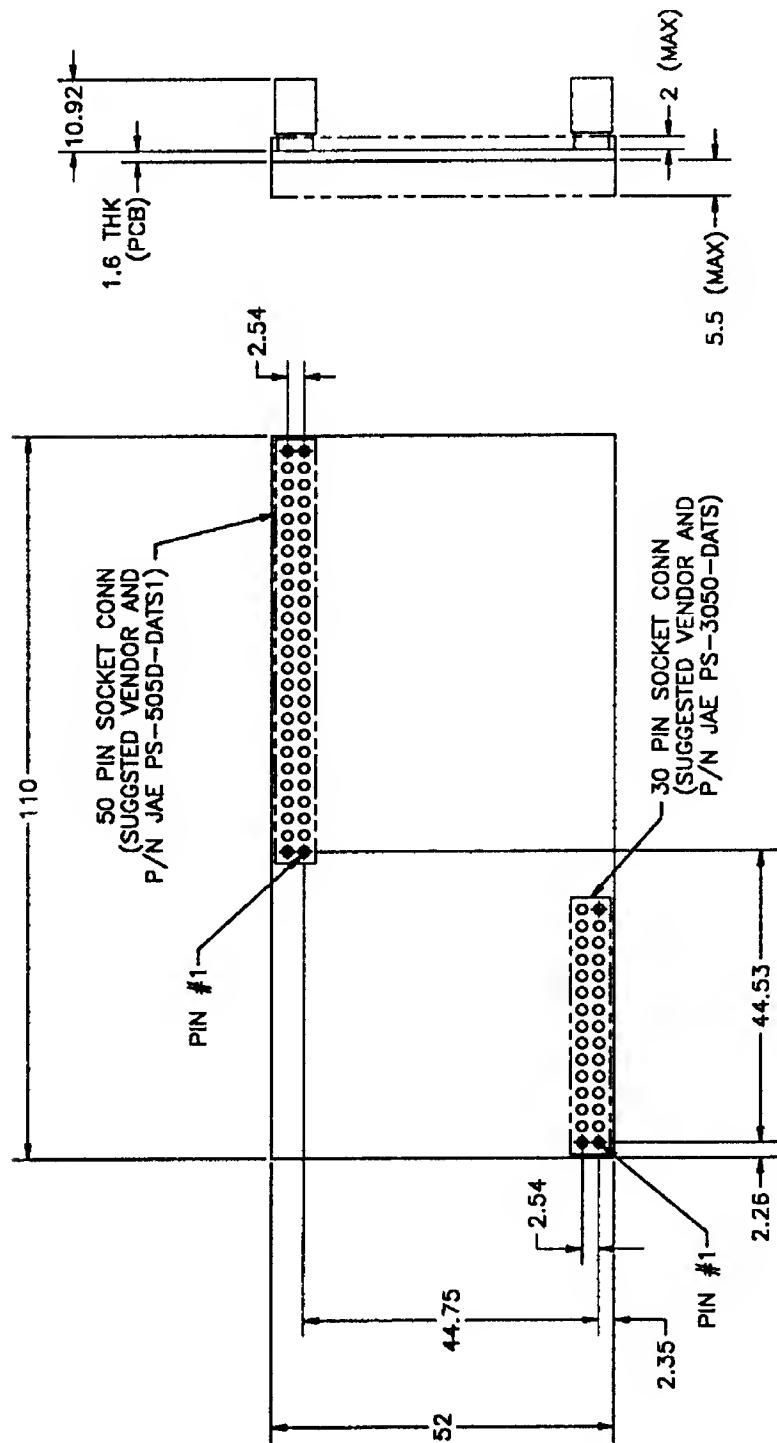
The internal expansion port essentially includes a 68000 direct microprocessor interface. Since the Atari Falcon030 uses a 68030 microprocessor there are some important differences from the 68000 bus. In particular, signals such as UDS, LDS, AS, and DTACK have been synthesized from the 68030 equivalents. In addition, the expansion bus has 16 bit data and 24 bit address busses.

No signal should ever be connected to more than one equivalent TTL load. Failure to follow this guideline will cause the system to become unreliable or fail completely.

Microprocessor Bus Signals

A(23:1)	Lower 23 bits of 68030 address bus
D(15:0)	Upper 16 bits of 68030 data bus (D(31:16))
UDS, LDS	Data Strobes (68000 compatible)
AS	Address strobe
DTACK	Data Transfer Acknowledge
RXW	Read/Write
FC(2:0)	Function code (68030 compatible)
RESET	Reset (active low)
HALT	CPU Halt

Bus Arbitration Signals

BR	Wire-Or'ed, active low bus request
BGK	Wire-Or'ed, active low bus grant acknowledge
BG	Daisy chained, bus grant
CPUBGI	Bus grant in, direct from CPU
CPUBGO	Bus grant out, to lower priority devices

The signals BR and BGK are wire or'ed together with every other alternate bus master in the system. The bus masters are:

```
    Top Priority↓ 68030 CPU
               ↓ Expansion (optional)
               ↓ DMA (For SCSI and Floppy disk drive)
               ↓ Sound Record
               ↓ Sound Playback
               ↓ BLiTTER
  Bottom Priority↓ Expansion
```

Expansion port devices can choose where they sit in bus priority. By using CPUBGI and CPUBGO they will have priority just below the CPU, but above DMA. Using BG, they will have lowest priority, just below the BLiTTER. Cards which do not use CPUBGI and CPUBGO, must connect these two signals together. If no card is installed, a jumper connects these signals.

Devices sitting at the top of the bus arbitration chain are intended to be µprocessors or other devices that are capable of relinquishing to other devices within one or two bus cycles. If an expansion board wishes to sit at the top of the chain it must guarantee a maximum response time of 1 microsecond to maintain system integrity. The worst case device is currently the floppy disk. If the DMA channel cannot empty its FIFO in time a sector of data will be lost. (SCSI does not have this problem since SCSI devices are by their nature buffered). Excessive response times may also cause Sound DMA to lose words when running in continuous mode.

To request the bus, a peripheral should pull BR low (with an open collector output), wait for BG to go low, and then acknowledge by pulling BGK low (again, with an open collector output). The conditions under which BGK can be pulled low can be somewhat complex since there are multiple alternate bus masters. Designers are urged to consult the 68030 documentation for a complete description.

Interrupt Signals

EINT1   Active high, level 1 interrupt
EINT3   Active high, level 3 interrupt

MFP_IEI   Active low, MFP (level 6) interrupt enable
MFP_INT Active low, Wire-Ored, level 6 interrupt
IACK   Active low, level 6 interrupt acknowledge

IPL(2:0)   Active low, CPU interrupt priority level
         indicators

EINT1 and EINT3 allow peripherals to interrupt at levels 1 and 3 respectively. These signals are decoded and prioritized by custom logic to generate a processor interrupt. MFP_INT can be used in conjunction with IACK and MFP_IEI to generate a high priority level 6 interrupt. The peripheral is positioned at a higher priority than the MFP or DSP (which can also cause level 6 interrupts). Peripherals should pull MFP_INT low (with an open collector output) while holding MFP_IEI high to hold off the MFP from asserting its own interrupt vector. When IACK goes low together with LDS, the peripheral should put a vector onto the data bus.

The IPL(2:0) signals must not be driven by peripherals since they are internally driven by custom logic. They are only included for devices which may want to monitor these signals.

Clock Signals

CPUCLK Set to 8MHz at reset, then set to 16MHz by
       TOS. This clock is used by the system bus
   to synchronize all bus cycles
500KHZ   500KHz fixed clock

Neither of these clocks should be loaded with more than one TTL type device (or equivalent) under any circumstances. Excessive loading of these clocks (or any other signals on the expansion bus) will lead to system unreliability or failure.

Bus Access

Slave Devices/RAM: The address (A23-A1) and the functions codes (FC2-FC0) along with AS must be used for decoding. Devices that require more than 4 CPUCLKs (i.e., DTACK is not generated before S5) must activate EXPAND by the end of S3. This allows 1 CPUCLK (62.5ns) from AS until EXPAND must be valid. It must be held until AS returns high. EXPAND is a wire-or'ed signal and must be driven with an open collector output. It can only be driven low, if AS is low. EXPAND can only be driven for address spaces that do not conflict with system devices and system RAM.

Bus Masters: For proper operation Bus Masters should emulate the 68030 timing if BMODE is pulled low. If BMODE is high, then the Bus Master is emulating a 68000 interface. The system control logic uses the BMODE signal to determine which edge of the CPUCLK to sample AS on. BMODE can only change state by an alternate Bus Master when it owns the bus. An alternate Bus Master will own the bus if it won arbitration for the bus and then AS is sampled inactive on two consecutive rising edges of CPUCLK. BMODE must remain valid for the entire bus cycle and be stable before AS is active.

Memory Map: Peripheral devices can use addresses in the range F10000 to F9FFFF (576 Kbytes) and any of the RAM space which is not occupied by RAM (address below E00000) and EXPAND.

Bus Interface Timing For EXPAND

Bus Arbitration
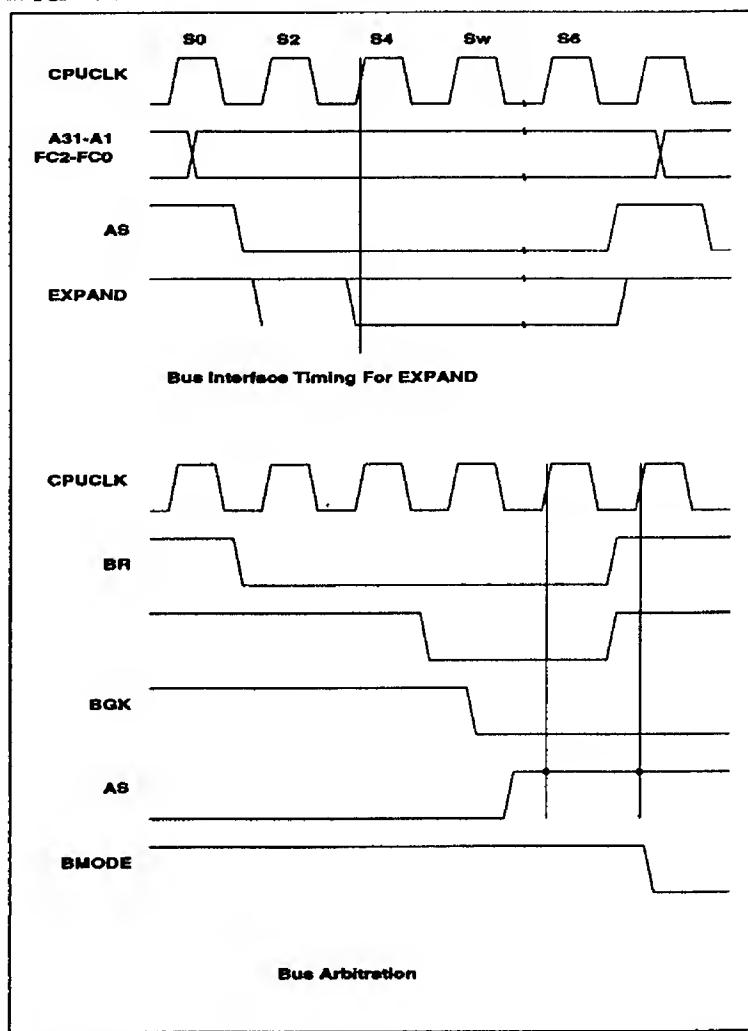
Video Port

The Atari Falcon030 has a new video port connector. This connector contains all the signals necessary for connection to an analog VGA monitor as well as an ST or STE compatible color or monochrome monitor. In addition, it includes the signals necessary for external GENLOCK devices including an external video dot clock, and insertion of external Vsync. The Atari Falcon030 video connector is a DB19 male. Its pinout is as follows:

Pin#	Signal	Pin#	Signal
1	Red	11	GND
2	Green	12	Composite video / Composite Sync
3	Blue	13	Hsync
4	Mono/Overlay	14	Vsync
5	GND	15	External clock input
6	Red GND	16	Even-Odd
7	Green GND	17	+12V
8	Blue GND	18	M1
9	Audio out	19	M0
10	GND		

Pin 4. Mono/Overlay

This pin is a one bit monochrome video output when in ST-High resolution (640 x 400). It has levels compatible with the ST, STE and MegaSTe.

In True color mode this pin represents the same polarity as bit 5 (the overlay bit) of each pixel:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	R	R	R	R	R	G	G	G	G	G	X	B	B	B	B	B

For standardization, we have defined this bit as follows:

Bit 5	Pin 4	Meaning
0	Low	Transparent (external video)
1	High	Overlay (Atari Falcon030 video)

The overlay bit becomes active one pixel clock period before analogue RGB:

```
                                              External Clock
 __    __    __    __    __    __    __    __    __
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
   |__|  |__|  |__|  |__|  |__|  |__|  |__|  |__|  |__

                                              Pixel Clock
 _____       _____       _____       _____       __
|     |     |     |     |     |     |     |     |
      |_____|     |_____|     |_____|     |_____|

                                              RGB (Pins 1, 2, 3)
 ____      ____      ____      ____      ____
     \    /    \    /    \    /    \    /
  0   \  /  1   \  /  2   \  /  3   \  /
      /  \      /  \      /  \      /  \

                                              Overlay (Pin 4)
 ____      ____      ____      ____      ____
     \    /    \    /    \    /    \    /
  1   \  /  2   \  /  3   \  /  4
      /  \      /  \      /  \

       ----->  T1  <-----
```

	Min	Typ	Max
T1	4ns	9ns	20ns
R,G,B, Propagation Delay		12ns	24ns
Analog Settling Time			14ns
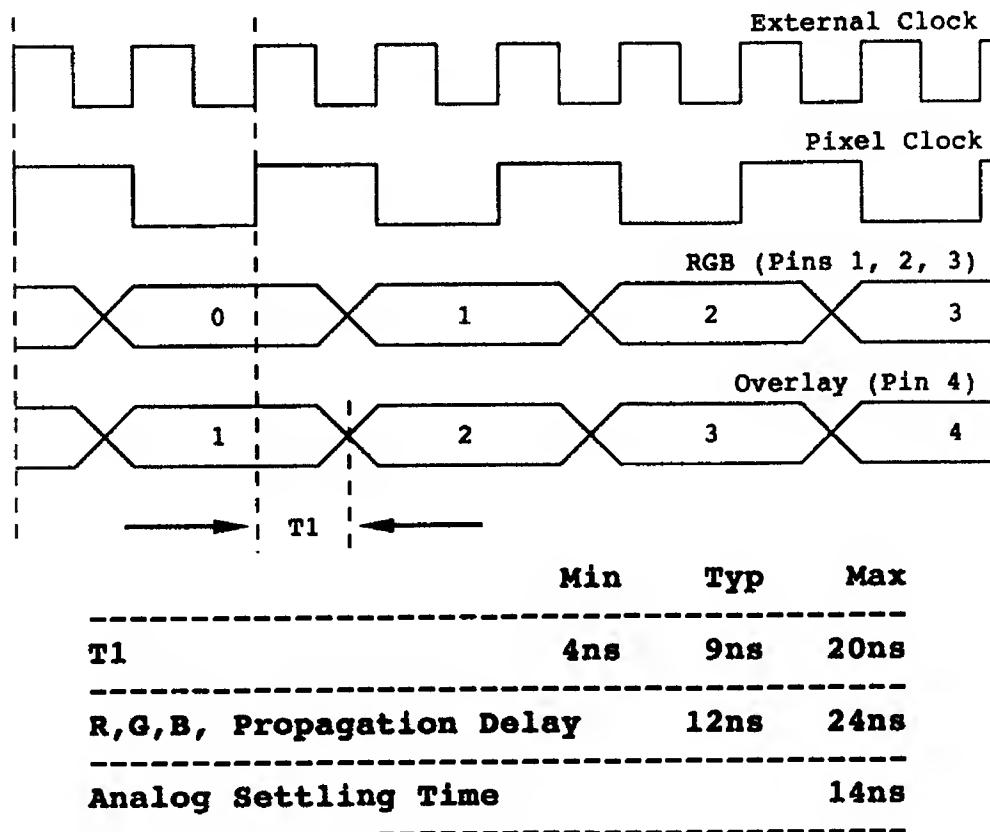
Note that the externally supplied clock (Pin 15) can be one, two or four times the frequency of the actual pixel clock used.

Typically this feature will be used to select between the Atari Falcon030 and externally generated video on a pixel by pixel basis. It could be called a one bit chroma-key, useful for overlays and video titling.

Note that the overlay bit is undefined outside of the raster data area. It is expected that most applications of the overlay bit will be running in overscan modes where only the data area is visible.

Pin 9. Audio out

This signal represents the same signal that goes to the internal speaker except that it cannot be disabled. It has a level of 1.4V RMS.

Pin 12. Composite Sync / Composite Video

On Peritel machines, this pin is Composite Sync. On all other machines, this pin is Composite Video.

Pin 14. Vsync

This pin can be programmed as an input to the Atari Falcon030. When it is an input, a low level on Vsync will hold the vertical timing generator in a reset condition. This feature is typically used by external Genlocking devices.

Hsync should not be programmed as an input. Horizontal locking is achieved with a phase locked loop, controlling the external video clock (pin 15). To avoid contention at reset time, a resistor should be used in series with the external Vsync.

Pin 15. External clock input

An external video source can drive a clock input into this pin synchronous with the external video dot-clock. The Atari Falcon030 will use this signal as master video clock, when selected in software.

Internally, this signal is padded with a 68Ω resistor and then pulled high with a 4.7k resistor. This signal should be driven by a 74HCxx or 74HCTxx type device, with a 50/50 duty cycle clock between ground and +5V. The maximum frequency this input can be driven at is 32MHz.

Pin 16. Even-Odd

In interlaced modes, this signal is low on even frames, high on odd frames.

Pin 17. +12V

This voltage level is necessary for Peritel interfaces. Peripherals can draw up to 100mA on this pin. It is internally fused.

### Pins 18,19. Monitor select 1,0

These pins are internally pulled high and are read by the operating system to determine the type of monitor

connected. The operating system then uses this information
to set up video timing values suitable for that particular
monitor.

The values assigned are as follows (1 -> +5V, 0 -> Gnd):

```
M1   M0   Monitor type
------------------------------
 0    0   ST Monochrome
 0    1   ST Color
 1    0   VGA
 1    1   TV
------------------------------
```

Composite Video Genlock

VGA Genlock

In from Reference

V H R G B

Vsync ——————— 200

Hsync —————→ | Phase Detector |

Extclock ←— | VCO | ←— | ≈ |

B —→
G —→
R —→

Overlay —————→

Video
Switch
Pixel
Rate

Hsync ——————
Vsync ——————

V H R G B

Out to Monitor
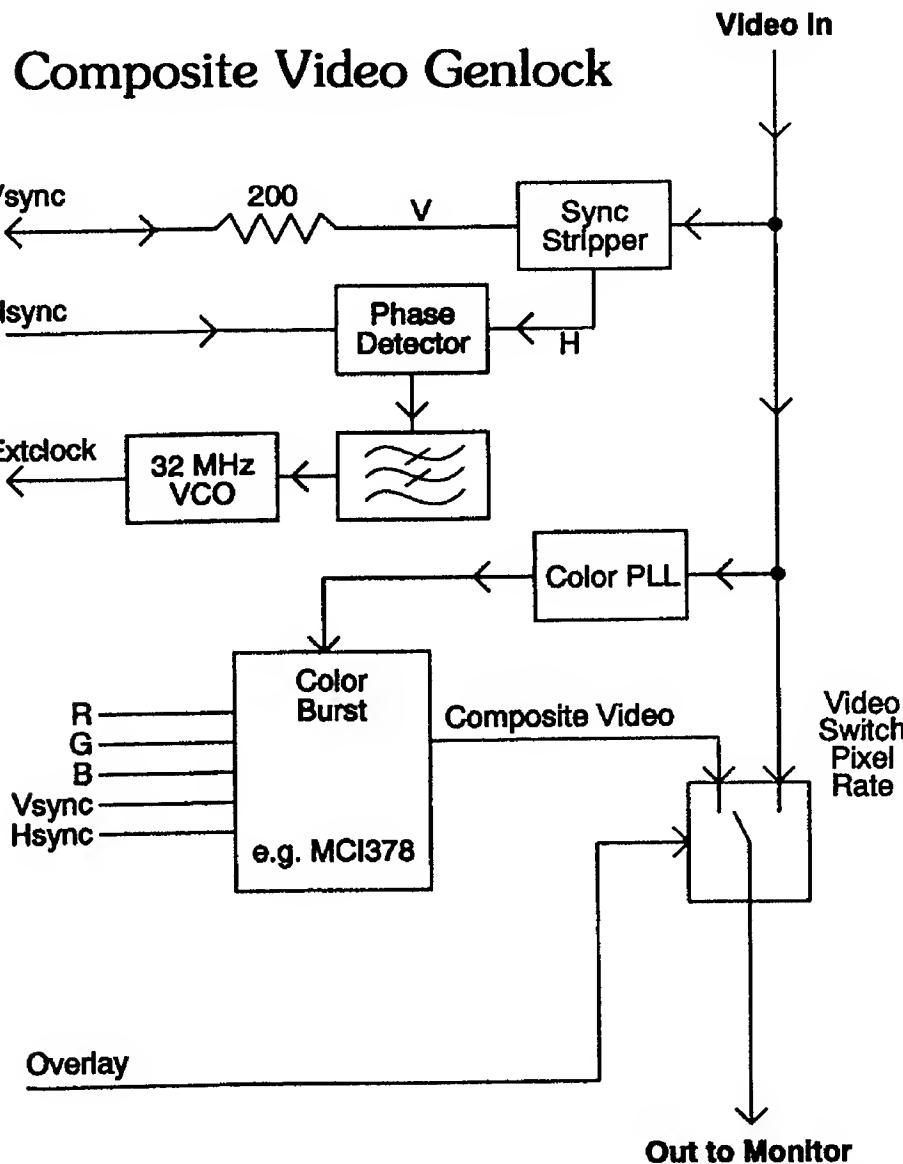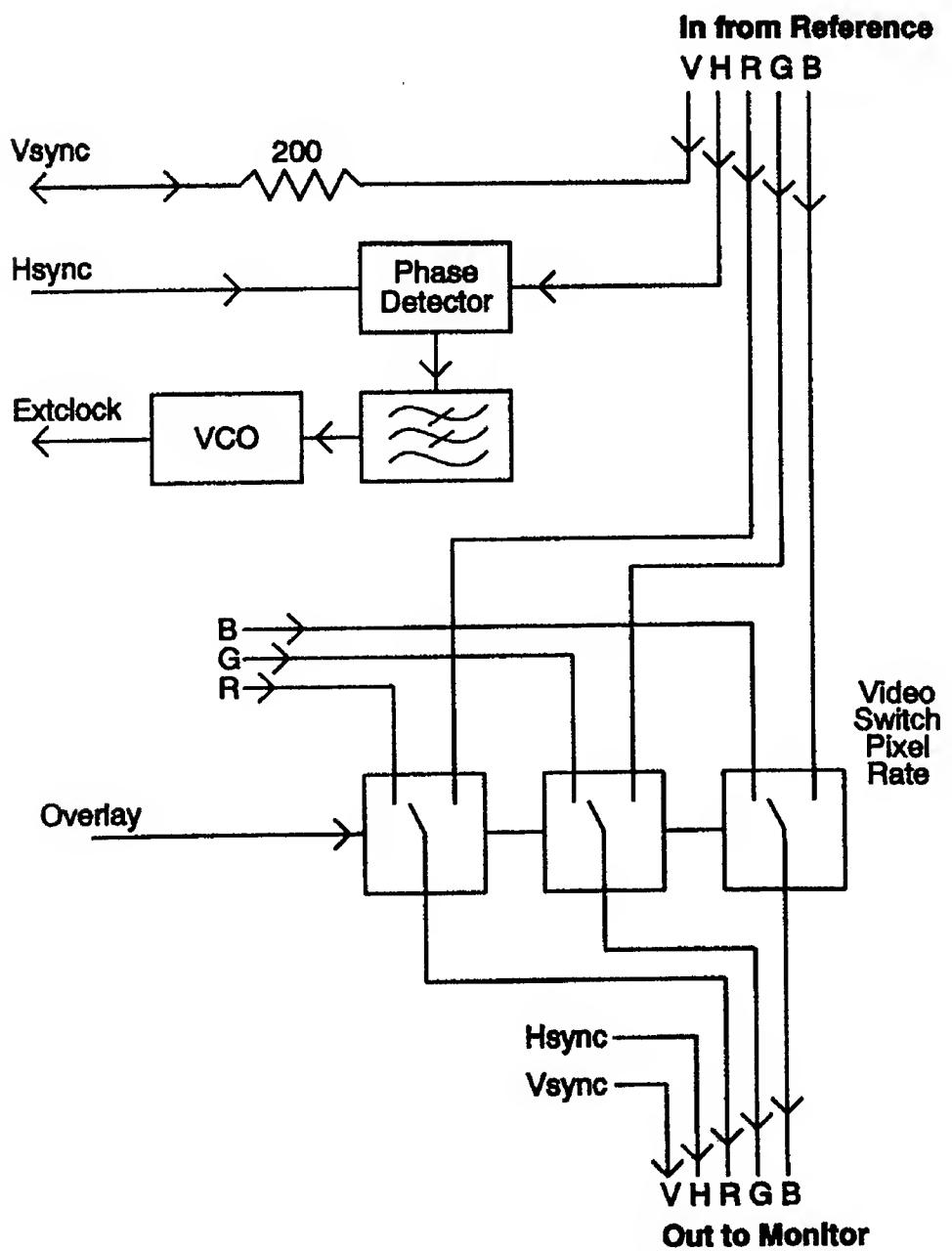
Digital Signal Processor (DSP)
and Audio Subsystem

Overview

The Atari Falcon030 contains a sophisticated digital processing and audio sub-system...

32 MHz 56001 Digital Signal Processor with 96K bytes of zero wait-state SRAM.

Eight track, 16-bit digital DMA record channel.

Eight track, 16-bit digital DMA playback channel (operating in parallel with digital record).

On-board 16-bit stereo DACs, feeding the internal loudspeaker and headphone jack.

On-board 16-bit stereo ADCs, and stereo microphone jack.

Sophisticated data path matrix between DSP, DMA, Codec and external connector.

Sample rates up to 50KHz.

Serial data transfer rates up to 1MByte per second.

Loudspeaker or headphones can monitor any stereo channel of 8 track digital playback data.

External serial record and playback channels connect to industry standard DACs, ADCs and S/PDIF components with minimum additional logic.

The block diagram on the following page describes the Digital processing sub-system.
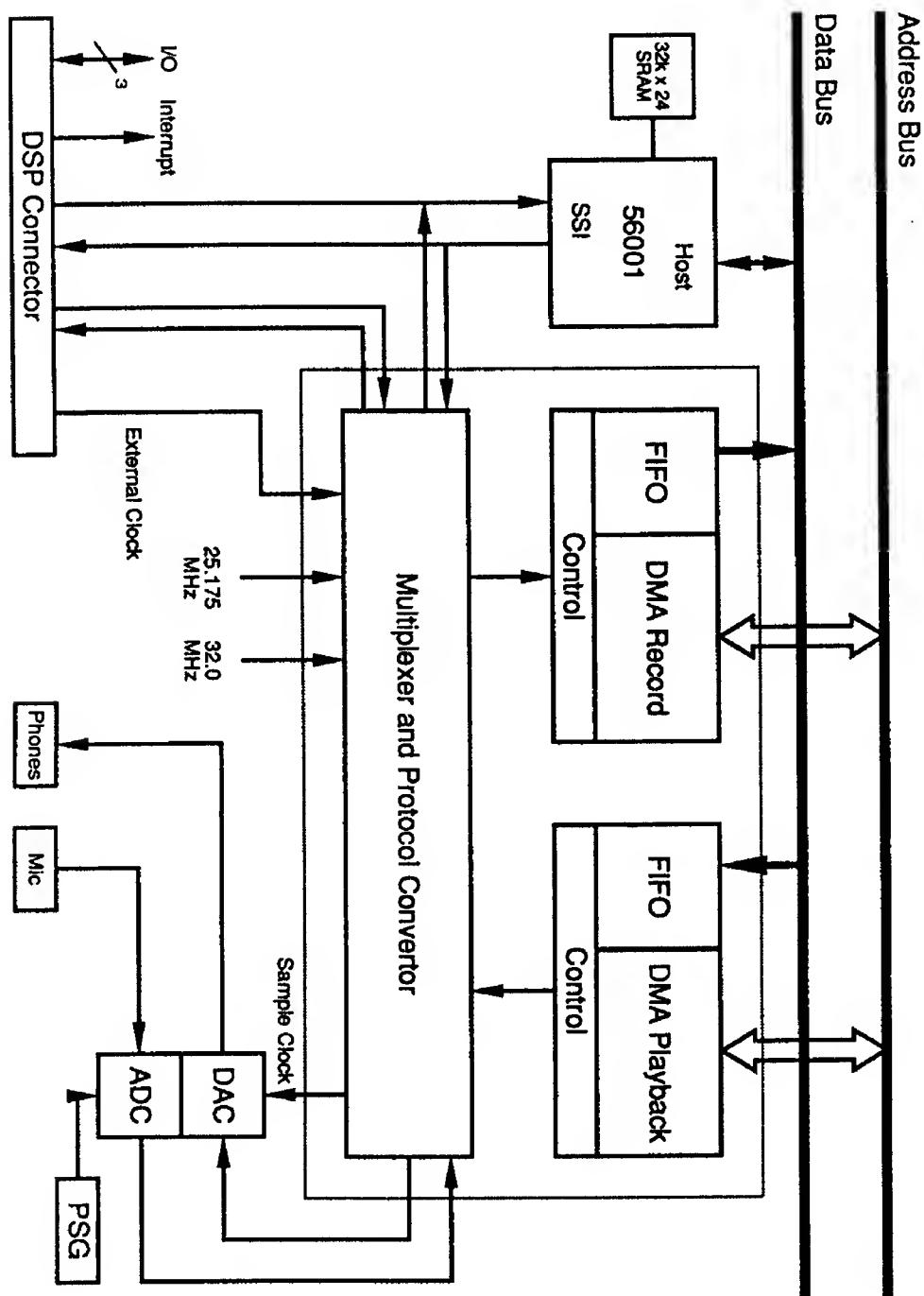
The digital processing sub-system has many features which make it ideal for audio processing. However, the data being processed can also be video (images), graphics objects (3-D image manipulation) or any other general purpose data.

To maintain the maximum flexibility, the Atari Falcon030 provides an extremely general connection system between these components. All data transfers are in a synchronous serial format. Any component can talk with any other. Since some of the components have real time response requirements, the clocking schemes have also been made especially general and flexible.

Communications

Any two devices in the sub-system can talk with each other. To allow them to talk you need to connect them together correctly. This requires several things:

1) Connect the two devices (a receiving device to a source device)
2) Select the source clock
3) Select the communication protocol (handshake or continuous)

Connections

There are four devices capable of sending data and four devices capable of receiving data. To allow any connection therefore requires a four by four matrix:

SOURCE DEVICE

```
EXTERNAL INPUT
   CHANNEL

DSP TRANSMIT

DMA PLAYBACK

   L Mic
PSG ──<        ADC
   R Mic
                +

              DAC
      Headphone   Speaker

         DSP      DMA      EXT.
       RECEIVE   RECORD   OUTPUT
                          CHANNEL

         RECEIVING DEVICE
```
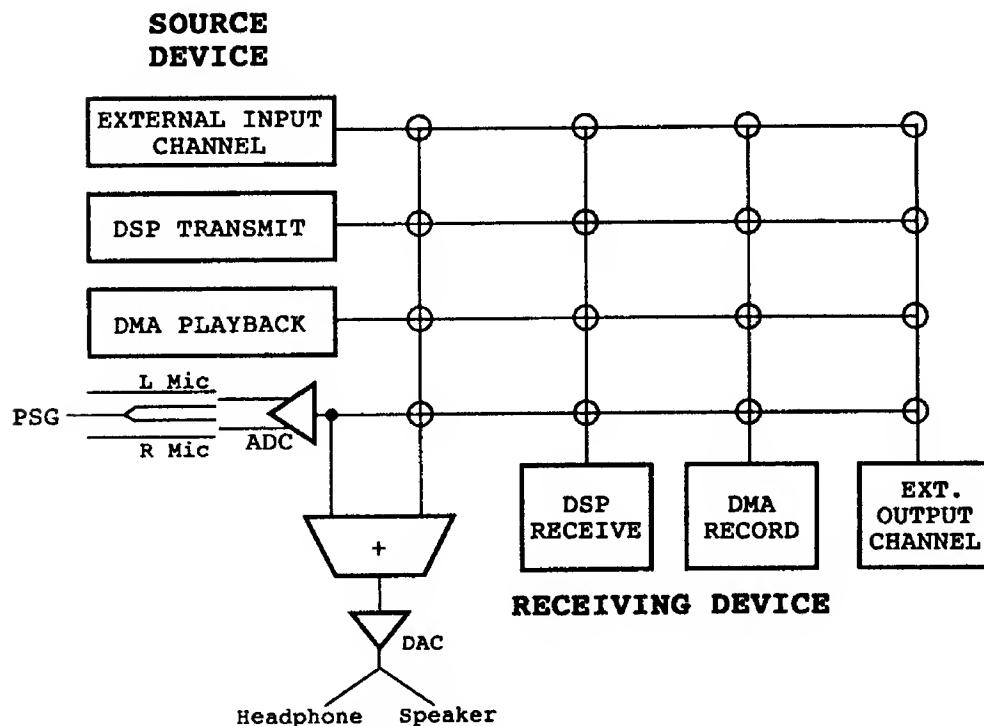
Each receiving device can have its data path connected to any one source device. Source devices "source" data. For example, the ADC represents data from the microphone jack so the ADC is a data source. It can send it's data to any (or all) receiving devices. See the "Devices" section for more details.

Clock Sources

All the data connections shown above, are actually serial data paths which include a bit clock, data, and synchronization signal.

There are three possible clock sources in the system:

Internal clock (25.175 MHz)
Internal clock (32 MHz)
External clock

Each source device must select one of these clocks as its master clock. The Codec can only use the Internal 25.175MHz, or External clock.

The bit clock is taken from the master clock divided by a programmable value of 4 to 24 (in increments of 4). The Sample rate is then the bit rate, divided by 128:

```
Master Clock      Divide    Bit Rate    Divide     Sample Rate
                    by                     by
            >       n       >            128        >
```
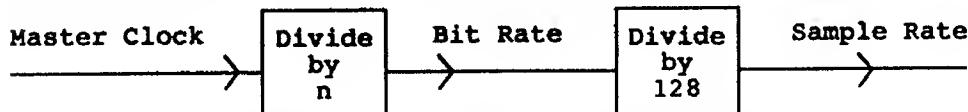
Since the bit rate is 128 times the sample rate, there is room for eight 16-bit samples per sample period.

Master clock	Divisor(n)	Bit Rate		Sample Rate		
25.175 MHz	4	6.29375	MHz	49.17	KHz	(50KHz)
22.5792 MHz	4	5.6448	MHz	44.1	KHz	(CD)
24.576 MHz	4	6.144	MHz	48.0	KHz	(DAT)
32.000 MHz	4	8.000	MHz	62.5	KHz	

The internal 25.175 MHz clock is used to support STE compatible 50KHz, 25KHz, and 12.5KHz sound sample rates. (Note that the built in DACs do not actually support a 6.25KHz sample rate)

The internal 32 MHz clock is useful since it can be used to provide an 8 MHz bit rate (or 1 Megabyte per second), which is the maximum transfer rate of the DSP SSI interface.

The external clock comes from the DSP connector. It can run up to 32 MHz. Some useful external clock rates are shown below:

> 22.5792 MHz gives CD rate of 44.1 KHz
> 24.576 MHz gives DAT rate of 48.0 KHz

Communication protocols

Data sometimes gets lost. We all do it. Even a piece of perfectly well designed hardware can do it.

The maximum data rate of the DMA record or playback channels is 1 Megabyte per second each. Since the FIFOs are 32 bytes deep each sound DMA channel will require bus access approximately every 32 microseconds.

Unfortunately, poorly written software can create situations where this access requirement is not met. A combination of other devices may lock out the bus from sound DMA, particularly, badly behaved expansion port devices and true color video.

If the data is sound data and it is not critical, then an occasional overrun or underrun may be acceptable. If the data is JPEG video, DSP object code, or any other non redundant data, then you will want to guarantee it is never mislaid.

For precisely this purpose our system includes a special handshaking mode which prevents overrun or underrun. When in handshaking mode, the data rate can be variable since timely bus access cannot be guaranteed. This also means that in handshaking mode there is no concept of a sample rate, or left and right tracks, or multiple tracks at all. The data is simply transferred one word at a time as quickly as the source and receiving devices can communicate.

If timely bus access can be guaranteed it is better to use continuous mode. Continuous mode should be used for any real time applications (such as sound playback or record), and it will generally be more efficient for the DSP since its interrupt routines can be faster.

Devices

There are a total of four devices in the audio sub-system, each of which are full duplex. In other words, we actually have four data sources and four data receivers:

```
Device        Data Source       Data Receiver
----------------------------------------------
DMA           DMA Playback      DMA Record
Codec         ADC               DAC
DSP           DSP Transmit      DSP Receive
External      External Input    External Output
```

These devices can be connected together in a very flexible manner (as shown in the matrix under "Connections" earlier in this section).

Each device has its own special characteristics, which are described below.

DMA Input

The DMA input channel provides a fast path to system memory. Briefly, it includes a 32 byte FIFO on the data path synchronized with a memory addressing module which can fill memory in a linear, continuous or looping mode. The maximum data transfer rate is about one Megabyte per second.

The data and clock signals to DMA input must be synchronized. Source devices can send data to DMA input in either handshaked or non-handshaked modes.

In handshaked mode DMA Input must be the clock source. It uses a gated clock technique to stop data transmission if its FIFO becomes full.

In non-handshaked mode, DMA input receives a clock from the sending device. When its FIFO becomes half full it will attempt to write it to memory. If it cannot get access to the system bus in time, data will overflow.

Non-handshaked mode to DMA input is provided simply because it puts less burden on the sending device. However, when using it the user must be careful to limit the data transfer rate to within system bus bandwidth limits.

DMA Output

The DMA output channel provides a fast data channel from system memory to sub-system devices. It also has its own 32 byte FIFO which helps ensure that it can keep up with the real time response required by certain devices (such as the Codec DACs).

Data transfers can be done in either handshaked or non-handshaked modes. In handshaked mode a gated clock technique is used together with a flag signal from the receiving device to prevent overruns or underruns.

Non-handshaked mode is normally used for communication with DACs or other real-time devices. If the system bus becomes overloaded for any reason with higher priority bus masters data may be lost in non-handshaked mode.

As usual, the receiving device must be using the same clocks and protocol as DMA output to ensure correct data transfer.

Digital Signal Processor (DSP)

The Atari Falcon030 includes a Motorola 56001 Digital Signal Processor. This part offers the following features:

32 MHz operation, yields 96 MOPS.

1024 point complex FFT can be done in 2.07 milliseconds.

24 bit internal and external data paths, yielding 144 dB
   dynamic range.

56 bit accumulators.

The following operations can be executed in parallel in one
   instruction cycle:

> 24 x 24 multiply
> 56 bit addition
> Two data moves
> Two address pointer updates
> Instruction prefetch

1024 x 24 bits of on chip RAM.

512 x 24 bits of on chip ROM used for Mu-Law, A-Law and
   four quadrant Sine wave table data.

DSP Memory Map

In addition to the on-chip RAM and ROMs there are 32K words of external, zero wait state SRAM.

The memory map is configured as follows:

Program space is one contiguous block of 32K words.
X and Y data space are each separate 16K word blocks.
Both X and Y can be accessed as blocks starting at 0 or 16K.
Program space physically overlaps both X and Y data spaces.

Note that since program space overlaps X and Y space DSP software must be careful to avoid having program and data memory corrupt each other. Note that X:0, X:16K and P:16K are the same physical RAM location, and that Y:0, Y:16K and P:0 are also at the same physical RAM location.

```
$ffff ┌──────────┐  ┌──────────┐  ┌──────────┐
      │          │  │          │  │          │
      │          │  │          │  │          │
      │ Reserved │  │ Reserved │  │ Reserved │
      │          │  │          │  │          │
      │          │  │          │  │          │
$7fff ├──────────┤  ├──────────┤  ├──────────┤──────────┐
      │   16 K   │  │   16 K   │  │          │ Overlaps │
      │  Shadow  │  │  Shadow  │  │   32 K   │ X memory │
$3fff ├──────────┤  ├──────────┤  │ Program  ├──────────┤
      │   16 K   │  │   16 K   │  │   RAM    │          │
      │ External │  │ External │  │          │ Overlaps │
      │   RAM    │  │   RAM    │  │          │ Y memory │
$01ff ├──────────┤  ├──────────┤  ├──────────┤──────────┘
      │ Internal │  │ Internal │  │ Internal │
      │ RAM/ROM  │  │ RAM/ROM  │  │   RAM    │
$0000 └──────────┘  └──────────┘  └──────────┘
        X Memory      Y Memory      P Memory
```
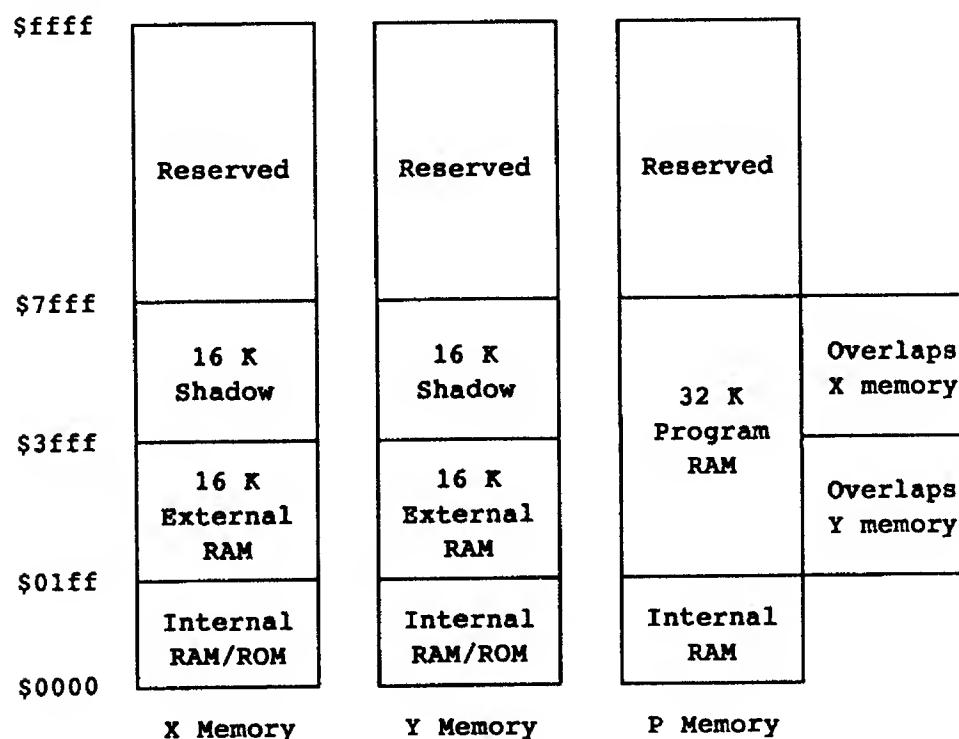
SSI Interface

The Atari Falcon030 brings out the six wire SSI port to the external DSP connector.

Host Port

Interface with the 68030 host is via the 56001 host port (port B). Data transfer by the host is via programmed I/O. In other words, the DSP host port appears in the 68030 memory map as eight byte locations. Data transfers by the host should always be conducted through the appropriate operating system calls (see the Atari Falcon030 software developer's guide).

DSP software transfers data to and from the host port in the usual way (see 56001 DSP User's Manual). The host can interrupt the DSP and vice-versa.

SCI

The 56001 three wire SCI port is not implemented in the Atari Falcon030. DSP software must not rely on the existence of any of the SCI registers, including the SCI timer, interrupts, or control and status registers.

Various versions of the Atari Falcon030 may or may not even include the SCI circuitry!

DSP expansion port

This DB26 female connector includes a variety of signals designed primarily for the connection of digital sound devices and modems. It can (and almost certainly will) be used for a number of other applications such as low cost laser printers, video digitizers, scanners and so forth.

The pinout is as follows:

DSP Connector, DB26, three row Female:

Pin#	Signal	Pin#	Signal	Pin#	Signal
1	GP0	10	GND	19	REC_DATA
2	GP2	11	SCO	20	R_CLK
3	GP1	12	SC1	21	R_SYNC
4	P_DATA	13	SC2	22	EXT_INT
5	P_CLK	14	GND	23	STD
6	P_SYNC	15	SRD	24	SCK
7	n/c	16	GND	25	GND
8	GND	17	+12V	26	EXCLK
9	+12V	18	GND		

Pin Description:

GP(2:0)	I/O	General purpose inputs and outputs. Can be individually set and read
EX_INT	I	General purpose interrupt input

SC0	I/O	DSP SSI port Pin SC0 (PC3), Receive clock
SC1	I/O	DSP SSI port Pin SC1 (PC4), Receive Sync
SC2	I/O	DSP SSI port Pin SC2 (PC5), Transmit Sync
SCK	I/O	DSP SSI port Pin SCK (PC6), Transmit clock
SRD	I/O	DSP SSI port Pin SRD (PC7), Receive Data
STD	I/O	DSP SSI port Pin STD (PC8), Transmit data

XO_DATA	O	External Serial Output, serial data
XO_CLK	O	External Serial Output, serial clock
XO_SYNC	I/O	External Serial Output, Sync

XI_DATA	I	External Serial Input, serial data
XI_CLK	O	External Serial Input, serial clock
XI_SYNC	I/O	External Serial input, Sync

EX_CLK	I	External master clock

+12V-	+12V power. Do not draw more than 300mA on this pin.

The signals on this port include several high speed clock and data lines. It is therefore essential that developers use correct drive and termination. In general, all signals should be terminated with a ferrite bead followed by a 68Ω resistor in series. This is the same type of termination used inside the Atari Falcon030 on all DSP port signals. A ferrite bead should be chosen that does not begin cutoff until 20MHz to 30MHz. Input signals from the peripheral should be driven by CMOS devices such as 74HCxx or 74HCTxx.

Total cable length should not exceed 24 inches and we strongly advise the use of twisted pair cables.

General purpose bits

Three bits are provided for general control purposes. They can be set, cleared or read as inputs through the operating system. At reset these three lines are programmed as outputs and driven low by TOS.

DSP SSI interface

These six pins are the SSI port from the Motorola 56001 DSP chip. The serial clock can operate up to one quarter of the 32 MHz DSP master clock rate, or 8MHz.

To use these pins to talk directly with the DSP you need to take care to avoid contention with the communication matrix by tri-stating the communication matrix outputs through the appropriate OS call.

External Serial Output channel

This three wire serial interface can be used to transfer data from the host computer. It can transfer data from the DSP, DMA playback channel, or on board analogue to digital convertor.

Data transfers use either continuous mode or a handshaked (gated clock) mode:

Signal	Continuous	Handshaked
XO_DATA	Output	Output
XO_CLK	Output	Output
XO_SYNC	Output	Input

In either mode, data changes on the rising edge of the clock. Data should be sampled on the falling edge of the clock.

In Continuous mode there are 128 clock cycles per sample period. XO_SYNC will go high for the first 16 bits of a sample period and then low for the remaining 112 bits. In each sample period a maximum of 8 tracks of 16 bit data can be transferred. Data words are transmitted MSB first, end-on-end, with no gaps in between them. The number of words per sample period is determined by the source device.

A typical sample is shown below:
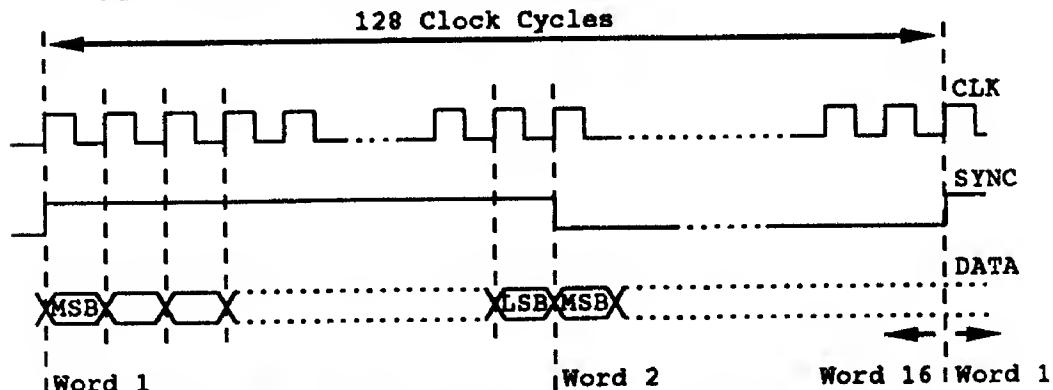
DATA and SYNC change on rising edges of CLK and should be
sampled on falling edges of CLK.

In Handshaked mode XO_SYNC becomes an input. The external device will pull XO_SYNC high, and if the source device is ready, XO_CLK will become active for 16 cycles (or one word) together with XO_DATA. XO_SYNC is sampled by the source device at the end of each word. If XO_SYNC is high and another word is ready to be sent, XO_CLK and XO_DATA will become active for another 16 cycles. A minimum of two clock periods will always be inserted between data words.

This gated clock technique will prevent overrun or underrun at either end of the data paths:
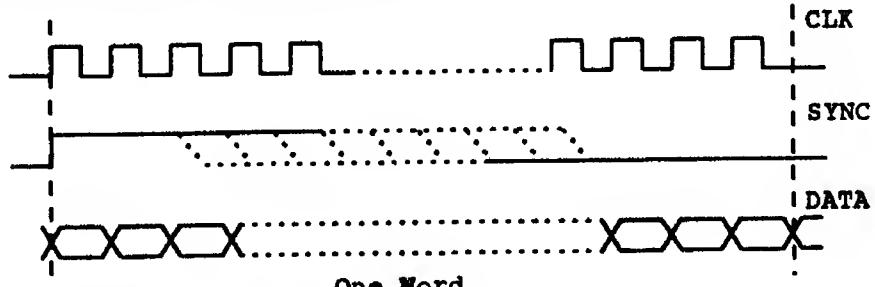
NOTE: SYNC hold time after first rising edge of CLK = 0ns

External Serial Input Channel

This three wire serial interface can be used to transfer data to the host computer. It can transfer data to the DSP, DMA record channel, or an on board digital to analogue convertor.

Data transfers use either continuous mode or a handshaked (gated clock) mode:

Signal	Continuous	Handshaked
XI_DATA	Input	Input
XI_CLK	Output	Output
XI_SYNC	Output	Input

In continuous mode it is the responsibility of the external device to synchronize to the XI_CLK and XI_SYNC outputs. Data should be changed on the rising edges of XI_CLK since it will be sampled on the falling edges. XI_SYNC will identify the start of a frame by going high for the first 16 clock cycles, and then low for the remaining 96 cycles.

In handshaked mode the protocol is basically the same as for the external serial output channel, except that XI_DATA is an input. When the external device has no data to send it must pull XI_SYNC low at least one clock cycle before the end of the previous sample.

External Master clock

This clock can optionally replace the internal 25.175MHz or 32.0MHz clocks. The maximum frequency allowable is 32 MHz.

CODEC

The Atari Falcon030 on board Codec is a high performance, 16 bit, stereo device. It includes a stereo DAC and stereo ADC.

16-bit Stereo DAC

The DAC output is directed to the on board loudspeaker (which can optionally be turned off), to the monitor port (for

monitors which have loudspeakers built in, such as the
SC1224), and the stereo headphone jack on the back panel.

DAC attenuation can be controlled for left and right channels
independently, through operating system calls.

Stereo Headphone Jack

The output port is a voltage drive with a peak voltage level
of 3V, and an RMS level of 2V. It is designed for a peak load
of 0.25W; this means that the load should have an impedance
greater than 32Ω.
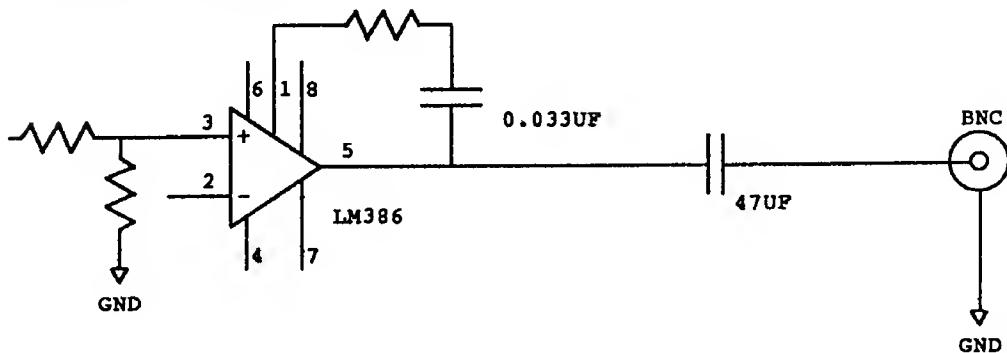
To help compensate for the poor low-frequency response of
headphones and small speakers, the headphone amplifier has
had a bass-boost circuit added to it which adds about 6dB to
the output level, centered at 100Hz, dropping to a 0dB boost
at 1KHz.

The power level present at the headphones is dependent on
the level in the input signal and the output impedance. If the
input (digital) value is assumed to be a 16-bit value scaled
between +/-1, then power level on the headphones is:

$$V_{OUT} = 3 * IN$$

$$P_{OUT} = (3 * IN)^2 / XH;$$

Where XH is the headphone impedance. For example, for
32Ω headphones the peak output power is:

$$P_{OUT} = 0.28 * (IN_{MAX})^2$$

The output is AC coupled by a 47µF capacitor. This means that there is a roll-off in the frequency response at low frequencies. The cut-off point can be approximated as:

$$F_{CUT\text{-}OFF} = 1/(2 * \pi * 47\mu F * XH);$$

Where XH is the impedance of the headphones. For example, with 32Ω headphones the cut-off is at 105Hz.

Note that the headphone output is a voltage. While the output is somewhat higher than normal line levels, output attenuation in the Codec can reduce this without loss of dynamic range. At the normal "line" impedance of 600Ω, the cut-off frequency will be lower; other internal limits keep the system to a cut-off of about 30Hz.

Internal Loudspeaker

The internal speaker is driven from a boosted op-amp. It is capable of output levels of 2V RMS (3.5V peak), and can drive loads as low as 8Ω. This means that the RMS output level is 0.5W. Peak levels will clip at 1.5W.
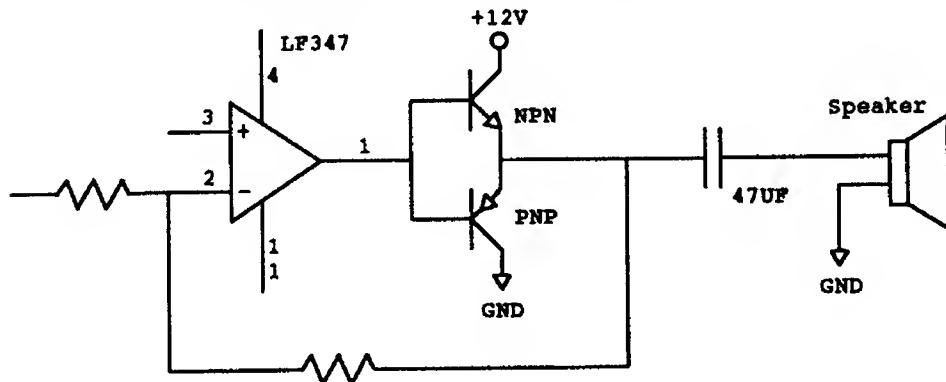
16-bit Stereo ADC

The ADC is connected to the microphone jack on the back panel. The ADC gain can be controlled through operating system calls. The PSG signals can optionally be fed to the ADC input.

Stereo Microphone Jack

The effective impedance of the microphone port is:

```
2.15K Ohm,   0    - 30Hz
1.77K Ohm,  30Hz - 900KHz
   0 Ohms   >900KHz
```

At DC, the input appears as a 2.2K resistor to +9V, and a 100K resistor to ground. The actual circuit used is shown below:
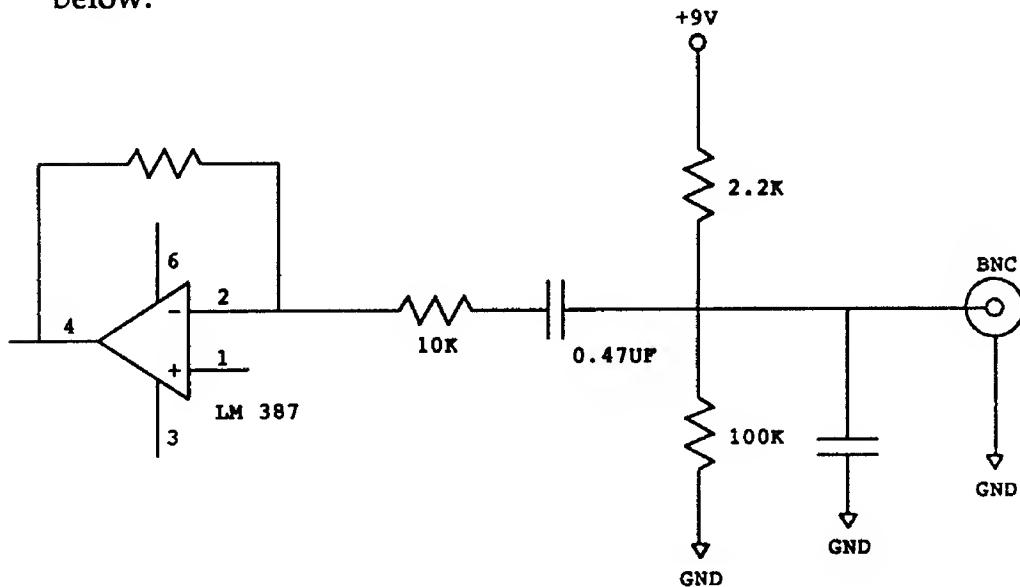
The maximum signal levels to be present at this port depend to some degree on the input gain set in the Codec. A "simple" formula is:

$$V_{MAX(RMS)} = (10^{-(0.075 * N)})/10;$$

where N is the value (0 to 15) of the input gain.

IMPORTANT! -- A 200k Ohm resistor should be used in series on each microphone input when connected to a 1V RMS "Line" level signal (such as the Line Out signals from a CD player).

Parallel Port

The Atari Falcon030 parallel port has been extended from previous TOS products, to include two additional signals - 'Acknowledge', and 'Select'.

The new parallel port now looks like this:

Parallel port. DB25, female.

Pin#	Signal	Pin#	Signal
1	Strobe	14	-
2	Data 0	15	-
3	Data 1	16	-
4	Data 2	17	Select
5	Data 3	18	GND
6	Data 4	19	GND
7	Data 5	20	GND
8	Data 6	21	GND
9	Data 7	22	GND
10	Acknowledge	23	GND
11	Busy	24	GND
12	-	25	GND
13	-		

'Acknowledge' is an input, active low from the printer. It is connected to the MFP pin GPIP1.

'Select' is an output, normally used to turn a printer on-line. It is connected to the PSG pin IOA3.

Serial port

The Atari Falcon030 serial port is connected to the 85c30 SCC
chip (rather than the 68901 MFP as in previous machines).
This is generally more powerful and flexible than the MFP.

Pin#	Signal		Input/Output
1	DCD	Carrier detect	i/p
2	RxD	Receive data	i/p
3	TxD	Transmit data	o/p
4	DTR	Data Terminal ready	o/p
5	GND	Ground	
6	DSR	Data set ready	i/p
7	RTS	Request to send	o/p
8	CTS	Clear to send	i/p
9	RI	Ring indicator	i/p

All signals are RS232 levels. Every signal except Ring
Indicator is connected to the appropriate 85c30 port B pin.

Ring Indicator is compatible with previous machines, and
connected to the MFP pin GPIP6.

Video Documentation

We recommend that all screen output be done via the GEM VDI. This technique allows an application to take advantage of higher resolutions and greater color capabilities of new screen modes yet still function in more limited situations. We do recognize, however, that direct screen output is something that applications authors are going to want to do. As a result we are documenting the screen memory organizations in all modes on the Atari Falcon030.

The 1, 2, 4 and 8 bit per pixel modes are arranged as they are in an ST, STE or TT. This organization consists of 16 bits of each plane in adjacent words until all planes are accounted for.

The 16 bit per pixel (true color) mode is organized as packed pixels. Each 16 bit word contains all of the information for a pixel.

Since this mode is a true color mode there is no palette to convert the data into RGB information for the video system. The information is encoded in each pixel where the 16 bits represent RRRRRGGGGGGBBBBB. An overlay mode exists where the 16 bits represent RRRRRGGGGGXBBBBB. The X bit is used as an overlay bit.

The video (_VDO) cookie is 0x00000300. This cookie is provided to developers so that applications that depend on the exact video specifications can do so. In general it is preferred for software to use the O.S. inquiry calls to check for specific abilities of the system.

OPCODE 5

WORD Setscreen(long log, long phys, WORD rez, WORD mode)

Setscreen() has been enhanced to handle the new Falcon video modes. If you pass a 3 in the 'rez' word and a modecode in the 'mode' word, Setscreen will set that mode ~~and realloc the screen RAM to match that mode.~~

[handwritten annotations]

Application programmers are better off using Setscreen() than VsetMode because Setscreen will handle reallocating the screen and will initialize the VDI for them. The VsetMode() call does NOT initialize the VDI with the new mode information.

[handwritten annotations]

However, VsetMode(-1) should still be used to inquire what resolution the machine is in before setting a new one. Then this information should be used to restore the previous resolution.

OPCODE 88

int Vsetmode(int modecode);

The Vsetmode(int modecode) call is used to place the Atari Falcon030 into a specific video mode. A bit-encoded value (called a "modecode") is passed to Vsetmode() to set the mode. Vsetmode() returns the previous mode that was set.

A "modecode" is a bit-encoded value that works as follows:

S	O	P	V	8	N	N	N

```
Low byte
  N » Bits per pixel ⎤  4 = 16 BPS    1 = 2 BPS
  N »               ⎬  3 =  8 BPS    0 = 1 BPS
  N »               ⎦  2 =  4 BPS
  8 » 80 column flag (if set, mode is 80 columns,
          otherwise it is a 40 Column mode)
  V » VGA flag, VGA monitor mode if set.. otherwise
          TV mode.
  P » PAL flag, PAL mode if set.. otherwise NTSC.
  O » Overscan flag / Multiplies both x and y by 1.2
          (Not used in VGA)
  S » ST compatibility flag.. It set, mode used will be
          ST compatible. (for ST Low, ST Medium, ST High)
```

X	X	X	X	X	X	X	F

```
High Byte
   F » Vertical flag. If set, Interlace mode used on a
         color monitor, double line used on VGA monitors.
   X » Reserved
```

A few modes are not allowed. 40 column 1 BPS modes are not supported. 80 column VGA 16 BPS modes are not supported.

To help make the building of modecode values easier, here is a table of defines:

```
#define   VERTFLAG   0x100
#define   STMODES    0x80
#define   OVERSCAN   0x40
#define   PAL        0x20
#define   VGA        0x10
#define   TV         0x0

#define   COL80      0x08
#define   COL40      0x0
#define   NUMCOLS    7

#define   BPS16      4
#define   BPS8       3
#define   BPS4       2
#define   BPS2       1
#define   BPS1       0
```

Using these defines, you can build a modecode for any possible mode. For example:

For True Color Overscan:
```
modecode = OVERSCAN|COL40|BPS16;
```

For ST Medium Compatibility mode on a Color Monitor/TV:
```
modecode = STMODES|COL80|BPS2;
```

For ST Low Compatibility mode in PAL on a Color Monitor/TV: `modecode = STMODES|PAL|COL80|BPS2;`

For 256 color, 80 column mode on a VGA monitor:
```
modecode = VGA|COL80|BPS8;
```

If you have a modecode and wish to know how many bits per pixel it has, use the following:

```
if(modecode & NUMCOLS) == BPS16)
    do_something_cool();    » You have true color mode «
```

The Vsetmode() call will return the previous modecode set. You must use this value to get back to whatever mode you were in before you made your Vsetmode call.

A word of warning: Vsetmode() does not provide error checking on valid modes. It will try to set modes that do not exist or that will not work on the monitor you are using. Be careful to set the proper mode for the right monitor!

The defines that are listed above as well as the xbios binding for Vsetmode() are defined in MODE.H on the distribution disk.

IMPORTANT NOTES: Vsetmode() does not adjust the video base address, allocate any memory for the new mode, or initialize the VDI. If you want to do these things, you should use Vsetscreen().

OPCODE 89

int mon_type(void)

The mon_type() function will return the kind of monitor that is currently in use. Here are the possible return values:

    0 = ST monochrome monitor
    1 = ST color monitor
    2 = VGA monitor
    3 = Television.

OPCODE 91

long VgetSize(WORD mode)

Returns the size of "mode" screen in bytes. Useful for easily determining the size of buffers to malloc for a given screen size.

OPCODE 90

void VsetSync(WORD external)

This will tell the VTG hardware whether or not to use external sync. The parameter 'external' is a bit value defined as:

```
00000hvc
        ^  external clock
     v- use external vertical sync
     h-- use external horizontal sync
```

This call only works in Falcon modes, not in compatibility modes or any four color modes.

OPCODE 93

void VsetRGB(WORD index, WORD count,
long *array)

Set colors by RGB value starting at "index" for "count" number of times. The RGB value is stored in the array. This code is called by vs_color() from the VDI. The format for the array is "xRGB" where x is not used.

This call is designed primarily for applications (i.e. games) that need to set large sections of the palette or perhaps the entire palette at once. If you need to set an individual color, you should use the VDI vs_color() call.

OPCODE 94

void VgetRGB(word index, WORD count,
long *array)

Get colors from the palette starting at "index" running until "count". Values are stored in the "array". The format of the values in array is "xRGB" and x means not used. Again, applications would be better off using the VDI to read or set colors (vq_color).

Like VsetRGB(), this call is designed primarily for the use of application programmers who need to set large banks of the palette at once.

OPCODE 150

VsetMask (ormask, andmask, overlay)

LONG ~~WORD~~ ormask, andmask; ~~overlay;~~

WORD overlay;

VsetMask is used to set the mask values used by VDI to modify the color values computed for vs_color(). The vs_color() function converts its input to a 16-bit RGB color value which is bitwise OR'ed with 'ormask' and then bitwise AND'ed with 'andmask'. This allows the application to set any color to be transparent (or not) in the 15 bit per pixel true color ~~overlay mode. This call may only be used in true color modes.~~ *modes with genlock and overlay.*

The default mask values are andmask=0xFFFF, ormask=0x0000. This combination has no effect. To set the overlay bit, use ormask=0x0020, andmask=0xffff. To clear the overlay bit, use ormask = 0x0000, andmask=0xffdf.

Now any color set with vs_color will have the overlay bit set.

parameter

If the 'overlay' ~~value~~ is non-zero, then the system will be put into overlay mode. If the 'overlay' ~~value~~ is zero, then the system will be taken out of overlay mode. *parameter*

Now any color set with vs_color will have the overlay bit cleared.

Sound Documentation
Low level Sound calls

The Atari Falcon030 _SND cookie is a bitmap of abilities.

Bit0 PSG	Bit3 DSP
Bit1 8-bit DMA	Bit4 Connection Matrix
Bit2 16-bit CODEC	

_SND = 0x3F

All of the calls return a long value even though only a portion of the long value maybe useable.

The following three examples illustrate some of the many possible ways that sound data might be laid out in memory:

```
     One 16-bit stereo track
|  L   |   R   |   L   |   R   |....
  WORD    WORD    WORD    WORD


     One 8-bit stereo track
|  L  |  R  |  L  |  R  |  L  |  R  |  L  |  R  |...
 BYTE  BYTE  BYTE  BYTE  BYTE  BYTE  BYTE  BYTE


     Four 16-bit stereo tracks
| TRK0,L| TRK0,R| TRK1,L| TRK1,R| TRK2,L| TRK2,R| TRK3,L| TRK3,R| TRK0,L| ...
   WORD    WORD    WORD    WORD    WORD    WORD    WORD    WORD    WORD
```
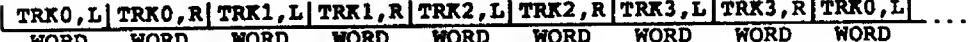
OPCODE 128

long locksnd();
Used as a semiphore to lock the sound system.

RETURNS: 1   Sound system is now locked.
                    SNDLOCKED (-129)

OPCODE 129

long unlocksnd();
Used to release the sound system for other applications to use.

RETURNS: 0   No Error.
                    SNDNOTLOCK (-128)

OPCODE 130

```
long soundcmd(mode,data);
```
This command is used to get or set the following sound
parameters. If a negative number is used as the input then the
current setting is returned.

MODE	OPERATION	MEANING
O	LTATTEN	Sets the left channel output Attenuation. Attenuation is measured in -1.5Db increments.
	INPUT:	(int) xxxx xxxx LLLL xxxx
		Where: LLLL- Attenuation to set.
		xxxx- Reserved.
	RETURNS	(int) xxxx xxxx LLLL xxxx
		Where: LLLL - Left Attenuation.
1	RTATTEN	Sets the right channel output Attenuation. Attenuation is measured in -1.5Db increments.
	INPUT:	(int) xxxx xxxx RRRR xxxx
		Where: RRRR- Attenuation to set.
		xxxx- Reserved.
	RETURNS	(int) xxxx xxxx RRRR xxxx
		Where: RRRR - Right Attenuation.
2	LTGAIN	Sets the left channel input gain. Gain is measured in 1.5Db increments.
	INPUT:	(int) xxxx xxxx LLLL xxxx
		Where: LLLL- Gain to set.
		xxxx- Reserved.
	RETURNS	(int) xxxx xxxx LLLL xxxx
		Where: LLLL - Left Gain.
3	RTGAIN	Sets the right channel input gain. Gain is measured in 1.5Db increments.
	INPUT:	(int) xxxx xxxx RRRR xxxx
		Where: RRRR- Gain to set.
		xxxx- Reserved.
	RETURNS	(int) xxxx xxxx RRRR xxxx
		Where: RRRR - Right Gain.

| 4 | **ADDERIN** | Set the 16 bit signed adder to receive it's input from the ADC, Matrix or both. The input to this function is a bitmap where: |

```
BIT   7 6 5 4 3 2 1 0
      0 0 0 0 0 0 M A
```

| | **INPUT:** | (int) | Bit
0- (A) ADC
1- (M) Matrix |
| | **RETURNS:** | (int) | xxxx xxxx xxxx xxMA |

| 5 | **ADCINPUT** | Set the input to the ADC. The input can either be the left and right channel of the PSG or the left and right channel of the microphone. The input is a bit map where if the bit is (0) it is a microphone input, or if the bit is a (1) it is a PSG input. |

```
BIT   7  6  5  4  3  2  1  0
      0  0  0  0  0  0  L  R
```

| | **INPUT:** | (int) | Bit
0- Right channel input.
1- Left channel input. |
| | **RETURNS** | (int) | xxxx xxxx xxxx xxLR |

| 6 | **SETPRESCALE** | Used for compatability. This prescale value is used when the DEVCONNECT() internal prescale value is set to zero. |

| | **INPUT:** | (int) | 0- Invalid
1- Divide by 640
2- Divide by 320
3- Divide by 160 |
| | **RETURNS** | (int) | Current divisor value. |

OPCODE 131

long setbuffer(reg,begaddr,endaddr);
This function is used to set the play or record buffers. reg selects playback or record, while begaddr and endaddr are the buffers beginning and ending location. The ending address is the first invalid data location.

(int)   reg   - (0) Sets playback registers.
                      - (1) Sets record registers.
(long) begaddr - Sets the beginning address of the buffer.
(long) endaddr - Sets the ending address of the buffer.

RETURNS: 0   No Error.

OPCODE 132

long setmode(mode);
This function is used to set record or playback mode. The modes are as follows:

MODE	OPERATION
(int) 0	8 Bit Stereo
(int) 1	16 Bit Stereo
(int) 2	8 Bit Mono

RETURNS: 0   No Error.

OPCODE 133

long settracks(playtracks,rectracks);
This function is used to sets the number of record or playback tracks. Note these are stereo tracks. When in 8-bit mono, two samples are read at a time.

(int)   playtracks (0-3)
(int)   rectracks (0-3)

RETURNS: 0   No Error.

OPCODE 134

long setmontracks(montrack);
This function is used to set the output of the internal speaker to one of the up to four tracks currently playing. The internal speaker is only capable of monitoring one track at a time.

(int)   montrack (0-3)

RETURNS: 0   No Error.

OPCODE 135

long setinterrupt(src_inter,cause);
This function is used to set which, if any interrupt that will occur at the end of a frame. If the frame repeat bit is on, this interrupt is used to allow for double buffering the playing or recording of sound. Interrupts can come from TimerA or the MFP i7.

| (int) | src_inter | (0) for timerA, (1) for MFP i7 |
| (int) | cause | (0) No interrupt, (1) Play, (2) Record, (3) Play or Record. |

RETURNS: 0   No Error.

OPCODE 136

long buffoper(mode);
This function is used to control the operation of the play or record buffers in the sound system. The input to this function is a bitmap. If mode is set to -1 then the current status of the buffer operation bits is returned.

(int)	mode	BIT	7	6	5	4	3	2	1	0
			0	0	0	0	RR	RE	PR	PE

Where:
RR - Record Repeat (1) on, (0) off
RE - Record Enable (1) on, (0) off

PR - Play Repeat (1) on, (0) off
PE - Play Enable (1)on, (1) off

NOTE: The sound system contains a 32 byte FIFO. When transferring data to the record buffer, software must check to see if the record enable (RE) bit was cleared by the hardware. If the bit was cleared then the FIFO is flushed, if not then software must flush the FIFO by clearing the record enable (RE) bit.

RETURNS: 0   No Error.
            or   Current setting of the buffer operation
                 bits.

OPCODE 137

long dsptristate(dspxmit,dsprec);
This function is used to tristate the DSP from the data matrix.

(int)   dspxmit   (0) Tristate, (1) Enable.
(int)   dsprec   (0) Tristate, (1) Enable.

RETURNS: 0   No Error.

OPCODE 138

long gpio(mode,data);
This is used to communicate over the General Purpose I/O pins on the DSP connector. Only the low order three bits are used. The rest are reserved. This call, depending on the mode, can be used to set the direction of the I/O bits, read the bits, or write the bits. At reset these three lines are programmed as outputs and driven low by TOS.

(int)   mode   (0) Set I/O direction (1) - read, (2) - write.
(int)   data   When setting I/O direction, a setting of
                 (1) indicates an output bit, where a (0)
                 indicates an input bit. A write operation
                 writes the data and a read operation
                 reads the current state of the GPIO port.

RETURNS: Value read for mode=1 otherwise 0

OPCODE 139

**long devconnect(src,dst,srcclk,prescale,
                  protocol);**

This function is used to attach a source device to any of the
destination devices in the matrix. Given a source device, this
call will attach that one source device to one or all of the
destination devices. This call also sets up the source clock
prescale value and protocol used.

(int)	src	Source device to connect to one or several destination devices. Source devices are: 3- ADC (Microphone/PSG) 2- EXTINP (External Input) 1- DSPXMIT (DSP transmit) 0- DMAPLAY (DMA Playback)
(int)	dst	A bitmap of destination devices that the source device will be connected too. 0x8- DAC (Headphone or Internal speaker) 0x4- EXTOUT (External out) 0x2- DSPRECV (DSP Receive) 0x1- DMAREC (DMA Record)
(int)	srclk	The clock the source device will use. There are three clock sources: 0- Internal 25.175MHz Clock 1- External Clock 2- Internal 32MHz Clock
(int)	prescale	Clock prescale. The sample rate is the clock value divided by 256, divided by the prescale value. These values are N-1 where N is the actual divisor. The range of N is from 1 to 12. N greater than 12 will result in a mute condition. The sndstatus() command can be used to reset the codec.

NOTE: If prescale=0 then the sound system uses the /1280,/640,/320,/160 compatability mode prescaler. See soundcmd().

(int)   protocol   Used to enable or disable handshaking protocol.
                        0- Enables handshaking
                        1- Disables handshaking

RETURNS: 0   No Error.

OPCODE 140

```
long sndstatus(reset);
```
This function gets the current status of the codec. The status is returned in the lower nibble (SSSS). Left (L) or Right (R) clipping is indicated if it has occured during the A/D conversion and filtering process.

(int)   reset   If one (1) resets the sound system. This is used to clear the overflow status bits if clipping has occured.

```
BIT 7 6 5 4 3 2 1 0
    0 0 L R S S S S
```

After this call the following conditions are set:

DSP is tristated.
Gain and attenuation is zeroed
Old matrix connections are reset
ADDERIN is disabled
Mode is set to 8 bit stereo (0)
Play and record tracks are set to track 0
Monitor track is set to zero.
Interrupts are disabled.
Buffer operation is disabled (0)

RETURNS: Status   0- No Error.
                       1- Invalid Control Field (Data still

assumed to be valid).

2- Invalid Sync format. This causes
   a mute condition.

3- Serial Clock out of valid range.
   This causes a mute condidion.

L- If (1) indicates left clipping is
   occurring.

R- If (1) indicates right clipping is
   occurring.

OPCODE 141

long buffptr(pointer);

This function returns the current position of the play and
record data buffer pointers. These pointers indicate where
the data is being read/written within the buffers themselves.
This function is also used to determine how much data has
been written to the record buffer. See buffoper().

(struct)   *pointer   A pointer to a structure of four longs
                        used to return the play and record
                        buffer pointers.

                        Structure
                        (long)Play buffer pointer.
                        (long)Record buffer pointer.
                        (long)Reserved.
                        (long)Reserved.

RETURNS: 0   No Error

Sample Rate Table

The following is a list of clock prescalers and their
approximate sample rates. Note that when setting the
internal codec source clock, only certain clock prescale rates
can be used. The 32Mhz clock can NOT be used by the codec
source clock. Also all clock rates marked with a (*) are
invalid clock prescale rates.

NOTE: If the devconnect() prescale is set to zero (0) then the
TT prescale divisor is used. If the devconnect() prescale is
zero (0) and the setprescale divisor is also set to zero (0) a
mute condition will occur. The setprescale divisor of / 1280 is
now invalid.

25.175 Mhz Prescale Table

Prescaler Value	NAME	Sample Rate
0		See (NOTE) above.
1	CLK50K	49170HZ
2	CLK33K	33880HZ
3	CLK25K	24585HZ
4	CLK20K	20770HZ
5	CLK16K	16490HZ
6*	14.285KHz	(invalid for codec)
7	CLK12K	12292HZ
8*	11.11KHz	(Invalid for codec)
9	CLK10K	9834HZ
10*	9.09KHz	(Invalid for codec)
11	CLK8K	8195HZ
12*	7.69KHz	(Invalid for codec)
13*	7.14KHz	(Invalid for codec)
14*	6.66KHz	(Invalid for codec)
15*	6.25KHz	(Invalid for codec)

Joystick/Keypad Matrix

The memory map that follows defines the joystick/keypad matrix. All of these inputs are read by scanning. You start the process by writing to FF9202 with the appropriate bit set low (all others set high). Then FF9200 and FF9202 are read to see if any bits are low. The button(s) pressed are read off of the matrix. As an example, FE is written to FF9202 and then FF9202 is read. Any low bits in FF9202 correspond to the first column in the table. Only controller 0 is treated in the table but the matrix for controller 1 is the same. Note that in the following, "ro" means when read and "wo" means when written.

```
FF9200  ----  ----  ----  xxxx   ro  BUTTON  (Button inputs)
  bit 0   controller 0 pin 6      Pause
  bit 1   controller 0 pin 10      F0    F1    F2    Option
  bit 2   controller 1 pin 6
  bit 3   controller 1 pin 10

FF9202  ----  ----  xxxx  xxxx   wo  JOY  (Joystick outputs)
  bit 0   controller 0 pin 4  ------------ X
  bit 1   controller 0 pin 3  -------------------- X
  bit 2   controller 0 pin 2  ---------------------------- X
  bit 3   controller 0 pin 1  ------------------------------------------ X
  bit 4   controller 1 pin 1
  bit 5   controller 1 pin 2
  bit 6   controller 1 pin 3
  bit 7   controller 1 pin 4

FF9202  xxxx  xxxx  ----  ----   ro  JOY  (Joystick inputs)
  bit 0   controller 0 pin 4
  bit 1   controller 0 pin 3
  bit 2   controller 0 pin 2
  bit 3   controller 0 pin 1
  bit 4   controller 1 pin 1
  bit 5   controller 1 pin 2
  bit 6   controller 1 pin 3
  bit 7   controller 1 pin 4
  bit 8   controller 0 pin 14      U    *    0    #
  bit 9   controller 0 pin 13      D    7    8    9
  bit 10  controller 0 pin 12      L    4    5    6
  bit 11  controller 0 pin 11      R    1    2    3
  bit 12  controller 1 pin 14
  bit 13  controller 1 pin 13
  bit 14  controller 1 pin 12
  bit 15  controller 1 pin 11
```

Controllers

Joysticks

FF9200 ▮▮▮▮▮▮▮▮▮▮▮▮▯▯▯▯
FF9202 ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯

Paddles

FF9210 ▮▮▮▮▮▮▮▯▯▯▯▯▯▯▯▯   (X Paddle 0)
FF9212 ▮▮▮▮▮▮▮▯▯▯▯▯▯▯▯▯   (Y Paddle 0)
FF9214 ▮▮▮▮▮▮▮▯▯▯▯▯▯▯▯▯   (X Paddle 1)
FF9216 ▮▮▮▮▮▮▮▯▯▯▯▯▯▯▯▯   (Y Paddle 1)

One pair of paddles can be plugged into Joystick 0 (Paddle 0).   A second set can be plugged into Joystick 1 (Paddle 1). The current position of each of the four paddles is reported at these locations. The fire buttons are the same as for the respective joystick. The triggers for the paddles are read as bits one and two of FF9202

Light Gun / Pen

FF9220 ▮▮▮▮▮▮▯▯▯▯▯▯▯▯▯▯   (X Position)
FF9222 ▮▮▮▮▮▮▯▯▯▯▯▯▯▯▯▯   (Y Position)

A light gun or pen can be plugged into Joystick 0. The current position that the gun or pen is pointing to is reported by these registers.
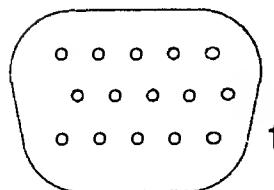
This pinout is for ports 0 and 1. Ports 2/3 are on the other DB15 connector.

1	UP 0	6	FIRE 0	11	UP 1
2	DN 0	7	VCC	12	DN 1
3	LT 0	8	NC	13	LT 1
4	RT 0	9	GND	14	RT 1
5	PAD 0Y	10	FIRE 1	15	PAD 0X

Atari DSP
Developer's Documentation
TOS Host Interface Routines

Communication between applications and the DSP on the
Atari Falcon030 must be done through a set of provided TOS
calls. This "virtualization" of the DSP hardware will insure
compatibility should the hardware be changed in future
machines.

DSP Memory Map

The private RAM that the DSP uses to store data or program
that will not fit into internal resources is supplied by three
32K Static RAMS. This memory appears to the DSP as
follows. Program space is one contiguous block of 32K
words. X and Y data space are each separate 16K blocks.
Both X and Y can be accessed, in the DSP's map, as blocks
starting at 0 or 16K. Program space physically overlaps both
X and Y data space so DSP software must take this into
account to avoid having program and data memory corrupt
each other. Note that X:0, X:16K and P:16K are the same
location in physical memory and that Y:0, Y:16K and P:0 are
also mapped to the same physical location. System services
will reside at the top of X memory along with DSP
subroutines. DSP subroutine BSS area will take up the top
256 words of both X and Y memory. A flush subroutine call
by the program will regain some of this memory back for the
program. As discussed in the next section, a Dsp_Available
call should always be made to determine the amount of free
ram on the DSP.

DSP Programs

Certain steps must be followed when programming for the
Atari platform. Some of the 32K words of DSP memory is
allocated for system tasks and resident subroutines and is
therefore not available for use by the DSP program. A host
process must therefore make a Dsp_Available call to find out
how much memory is left for its DSP program. If the amount

is satisfactory, the host process should reserve that memory area using a Dsp_Reserve call. This call will prevent the program's memory from being corrupted by the system. It is also necessary for the host process to prevent access to the DSP by another host process by making a Dsp_Lock call. This call must come before any other calls to manipulate the DSP. Doing this will insure that the status of the DSP will not be changed by someone else while the application is using it. When the host process is through using the DSP program it should do a Dsp_Unlock call to allow other processes to use the DSP. If a call to Dsp_Lock returns a "DSP busy" value, the host process should wait before making DSP system calls until a successful Dsp_Lock can take place. Failure to adhere to these rules will result in unpredictably bad results when communicating with the DSP. Before making an unlock call, the host application must make sure that its DSP process has restored the IPR (X:$FFFF) and MR to its original state.

DSP Subroutines

The existence of DSP subroutines allow the system to have multiple DSP processes resident at the same time. This saves the system the time of loading each program into the DSP every time it needs to be used. These subroutines will stay resident in the DSP until they are either pushed out by other subroutines or they are flushed out by a DSP program wanting more memory. DSP subroutines are subject to many more constraints and restrictions than are DSP programs. Subroutine code must be completely relocatable. When writing subroutine code, instructions should begin at address 0. When a subroutine is called through a host command, the subroutine can obtain it's starting PC through the host port. This beginning location which is sent by TOS should be read by the subroutine whether or not it is needed for relocation. Subroutine size is limited to 1024 DSP words of instructions. Anything larger would probably be more appropriately executed as a program. The code will be relocated somewhere into external DSP ram. Care should be taken to make any addresses used in the program (end addresses for do loops for example) relocatable based off of the original

program counter. Any initialized data must be declared
within the program space in which it is contained. A block of
X and Y memory has been set aside for a subroutines
undeclared variable space. This area is located in the highest
256 DSP words of memory in both the X and Y memory space
(X:3f00 - X:3fff). This area may be used freely by the
subroutine but since this area is used by all subroutines, it
should not be assumed that the memory will be preserved
the next time the subroutine executes. Host programs must
use the Dsp_Lock function before executing a DSP
subroutine. Since DSP subroutines are executed as interrupts
through host commands sent from the system, they need to
be terminated by an RTI after it has completed execution.
The subroutine should not assume any initial state of the DSP
since its state is determined by previously executed programs
and subroutines and not from a bootstrap. A typical
sequence of calls to execute a subroutine may look like the
following.

```
if(!Dsp_Lock())
{
    ability = Dsp_RequestUniqueAbility();
    handle = Dsp_LoadSubroutine(ptr,size,ability);
    status = Dsp_RunSubroutine(handle);
    Dsp_DoBlock(data_in,size_in,data_out,size_out);
    Dsp_Unlock();
}
```

A more efficient way of executing the subroutine would be to
first check to see if a subroutine already exists on the DSP
that would satisfy the applications requirements.

```
if(!Dsp_Lock())
{
    handle = Dsp_InqSubrAbility(ability);
    if(handle)
    {
        status = Dsp_RunSubroutine(handle);
        Dsp_DoBlock(data_in,size_in,data_out,size_out);
        Dsp_Unlock();
    }
}
```

Program Ability

A program's (and subroutine's) ability must be reported to the system when loading the DSP process. This ability is either a pre-defined ability which has been officially registered with Atari or a unique ability which was acquired by a Dsp_RequestUniqueAbility call. This ability can be used to determine whether the host needs to reload it's DSP process or whether it can use a process which already exists on board the DSP. The basic concept behind the host interface is that DSP programs and subroutines are not owned by the host application that loaded it. Once loaded, DSP programs become shared and freely usable by any host application that wants to use it.

DSP Library Functions
Data Transfer Routines

OPCODE 96

```
Dsp_DoBlock(data_in, size_in, data_out,
            size_out)
char *data_in;
long size_in;
char *data_out;
long size_out;
```

Dsp_DoBlock will handle block transfers of data between the host process and the process inside the DSP. Data pointed to by data_in will be passed to the DSP until size_in number of DSP words are transferred over (the number of bytes in a DSP word is returned by the Dsp_GetWordSize call). It is important to note that no handshaking will occur while the routine is feeding the data to the DSP. It will be assumed that for the purpose of this call, the DSP will be able to accept the data as fast as we can provide it. The call will wait for the first DSP word to be accepted by the DSP before beginning transfer of the rest of the buffer. After all of the data has been transferred to the DSP, Dsp_DoBlock will wait until the DSP has finished processing the data and is ready to send it back to the host (when the RXDF bit is set in the ISR register). At this time, size_out number of DSP words will be read from the DSP and stored into the buffer pointed to by data_out. Again, no polling of data ready bits will occur before data transfer. Also, we will read size_out number of words into the data_out buffer whether or not that much data actually exists for transfer from the DSP. If no data is expected out of the DSP, a zero should be placed in size_out. Similarly if no input is to be received by the DSP, size_in should be set to zero. Size_in and size_out are long values indicating the size of the arrays. Size_in and size_out are limited to a maximum of 64K.

OPCODE 97

```
Dsp_BlkHandShake(data_in, size_in,
                 data_out, size_out)
char *data_in;
long size_in;
char *data_out;
long size_out;
```

This call is identical to Dsp_DoBlock except that handshaking takes place during the transfer of the entire buffer. This call will be slower than Dsp_DoBlock and should only be used when the routine is expected to send/receive data faster than the DSP process can accept or send it. Size_in and size_out are long values indicating the size of the arrays. Size_in and size_out are limited to a maximum of 64K.

OPCODE 98

```
Dsp_BlkUnpacked(data_in, size_in,
                data_out, size_out)
long *data_in;
long size_in;
long *data_out;
long size_out;
```

Dsp_BlkUnpacked is another block transfer routine which works in a similar manner to Dsp_DoBlock. This routine will work only for TOS versions which return a value of 4 or smaller for Dsp_GetWordSize. Data_in and data_out are arrays of 32 bit long words. Size_in and size_out are the number of longwords in the array and the number of DSP words to transfer. Data is fetched from the least significant bytes of the longword and sent to the DSP. Similarly, data obtained from the DSP is placed into the least significant bytes of the size_out buffer. For example if Dsp_GetWordSize returned 3 (24 bits of DSP data). The least significant 24 bits of each longword would contain DSP data while the most significant 8 bits would contain something meaningless. (Note: These 8 bits are not guaranteed to contain zero. If the calling routine expects this byte to be cleared, it must mask it off itself). Size_in and size_out are

long values indicating the size of the arrays. Size_in and size_out are limited to a maximum of 64K.

OPCODE 123

```
Dsp_BlkWords(data_in, size_in, data_out,
             size_out)
long *data_in;
long size_in;
long *data_out;
long size_out;
```

Dsp_BlkWords takes blocks of signed 16 bit words and sends them to the DSP.   Words are sign extended before they are transferred. In a similar manner, Dsp_BlkWords takes the middle and low byte sent from the DSP and places them into the 16 bits of the output array. Data_in and Data_out are 16 bit integer arrays. Size_in and Size_out are long values indicating the size of the arrays. Size_in and size_out are limited to a maximum of 64K.

OPCODE 124

```
Dsp_BlkBytes(data_in, size_in, data_out,
             size_out)
long *data_in;
long size_in;
long *data_out;
long size_out;
```

Dsp_BlkBytes takes blocks of unsigned chars and sends them to the DSP. These character values are not sign extended before being transferred to the dsp. The low byte of the transfer register is placed into the character array during output to the host. Data_in and Data_out are 8 bit character arrays. Size_in and Size_out are long values indicating the size of the arrays. Size_in and size_out are limited to a maximum of 64K.

OPCODE 127

```
Dsp_MultBlocks(numsend, numreceive,
sendblocks, receiveblocks)
long numsend;
long numreceive;
struct dspblock sendblocks[];
struct dspblock receiveblocks[];

struct dspblock {
    int blocktype;  /*0= longs
                      1= signed 16 bit ints
2= unsigned chars*/
    long blocksize;
    long blockaddr;
} ;
```

Dsp_MultBlocks can be used to send multiple blocks of data
to and from the DSP while using only one trap call. Using
this call will save the overhead of making an XBIOS trap call
for every block that you want to send. The numsend and
numreceive parameters represent the number of dspblock
elements to expect in the input and output arrays.
Sendblocks and receiveblocks are the addresses of the two
dspblock arrays which contain the data to pass to and from
the dsp. A dspblock consists of a block type, a block size and
a block address. The block type lets the operating system
know what type of data is contained in the block (0 = longs,
1 = 16 bit signed ints, 2 = unsigned chars). The block size
indicates the number of elements in the block and the block
address is a pointer to the block of data.

OPCODE 99

```
Dsp_InStream(data_in, block_size,
             num_blocks, blocks_done)
char *data_in;
long block_size;
long num_blocks;
long *blocks_done;
```

Dsp_InStream will pass data to the DSP from the given
buffer via a DSP interrupt handler. Each time an interrupt

occurs telling the routine that the DSP is ready for more data, block_size DSP words will be transmitted to the DSP. As with the block move function, no handshaking will occur during this process. This routine will continue servicing interrupts until it has transferred over "num_blocks" number of blocks to the DSP. At that time the interrupt routine will tell the DSP to stop sending ready to receive interrupts. Dsp_InStream will update the long value pointed to by blocks_done to let the caller know how many blocks have been transferred over. The calling routine can periodically check this value to see if transmission has been completed. This routine allows the calling application to begin processing another batch of data as the current batch is being transferred to the DSP. As the routine's name implies, this call should be used instead of Dsp_DoBlock when a continuous stream of data is to be transmitted into the DSP. If on the other hand, a single large chunk of data needs to be transferred, it may be more efficient to use Dsp_DoBlock instead.

OPCODE 100

```
Dsp_OutStream(data_out, block_size,
                num_blocks, blocks_done)
char *data_out;
long block_size;
long num_blocks;
long *blocks_done;
```

Dsp_OutStream will fill the buffer pointed to by data_out via a DSP interrupt handler. The call is similar to Dsp_InStream above except that data is transferred from the DSP to the buffer at each interrupt. Again, block_size number of DSP words are transferred at each interrupt until num_blocks number of blocks has been transferred over. At that time, blocks_done will be equal to num_blocks informing the calling process that transmission has stopped.

OPCODE 101

```
Dsp_IOStream(data_in, data_out,
             block_insize,
             block_outsize,
             num_blocks,
             blocks_done);
char *data_in;
char *data_out;
long block_insize;
long block_outsize;
long num_blocks;
long *blocks_done;
```

Dsp_IOStream is a specialized form of the previously
documented stream handlers. This routine makes the
important assumption that every time a block of data is
ready to be transferred from the DSP to the host, the DSP
will at the same time be ready to accept as input another
block of data. By handling both the input to and output from
the DSP in one interrupt handler, the application can save the
overhead of servicing a second interrupt. When
Dsp_IOStream is first called, it "primes the pump" by
sending the first block of data to the DSP. It then installs an
interrupt handler to service "output is ready" interrupts from
the DSP. From that point on, each time an interrupt occurs,
the handler will fetch the block of data from the DSP and also
send a new block of data to the DSP. The variables which are
passed into the function are used in a manner similar to the
other stream processing functions. Data_in and data_out
represent the input and output buffers. Block_insize and
block_outsize represent the size of blocks in DSP words to
pass into and receive from the DSP. Num_blocks is the
number of blocks to transfer and blocks_done points to the
value which keeps track of the number of blocks which have
been transferred.

OPCODE 126

Dsp_SetVectors(receiver, transmitter)
void (*receiver)();
long (*transmitter)();

Dsp_SetVectors allows the host process to install a function which is called when an interrupt is received from the DSP. Receiver should point to a function that the user wants called when the DSP has sent data to the host process. Transmitter should point to the routine to be called when the DSP interrupts requesting data. If transmitter returns a non-zero long value, the XBIOS portion of the interrupt handler will send the low three bytes of the longword to the DSP. No data will be sent if the 32 bit long word which is returned is a 0. (To send back a 0 DSP word, OR in a value into the high byte of the returned value) If either receiver or transmitter are 0L, the corresponding interrupt will not be enabled. The host must remove its interrupts by using the Dsp_RemoveInterrupts call.

OPCODE 102

Dsp_RemoveInterrupts(mask);
int mask;

Dsp_RemoveInterrupts will stop the DSP from generating ready to receive or ready to send interrupts to the host. Mask is an 8 bit mask which represents the interrupt to turn off. 1 = No more interrupts when the DSP has data ready for the host; 2 = Don't generate interrupts when the DSP is ready to receive data from the host; 3 = Remove both types of interrupts. This call should be made if one of the previously described stream calls are made and a less than expected amount of data is passed to or from the DSP thereby not allowing the interrupt routine to terminate. It should also be used to remove interrupts installed by a Dsp_SetVectors Call.

OPCODE 103

```
size = Dsp_GetWordSize();
```

```
int size;
```

Dsp_GetWordSize returns the number of bytes which
represents a DSP word in the current system. It is important
for the application to use this routine to determine values
such as buffer size and block size. Buffer sizes for all of the
data transfer routines should be modulo the size returned by
this function. The value returned by this routine may
change in future versions of hardware.

Program Control Routines

OPCODE 104

state = Dsp_Lock()

Dsp_Lock should be called before making any other calls to the DSP Library. The call is intended to provide a way for host applications to tell whether or not the DSP is currently in use. A value of -1 returned by this function informs the calling application that a call to Dsp_Lock has already been made by another process. A return value of 0 means that the DSP is available and that you are free to make other DSP calls. The DSP will stay locked until a call to Dsp_Unlock is made.

OPCODE 105

Dsp_Unlock()

Dsp_Unlock should be used in conjunction with the Dsp_Lock call described above. A call to this routine tells the system that you are through with the DSP and that it is safe to allow someone else to begin using it.

OPCODE 106

Dsp_Available(xavailable, yavailable)
long *xavailable;
long *yavailable;

Dsp_Available returns to the calling process the amount of memory which is available to use in the DSP (See previous discussion on DSP memory map). Upon return from this call, the longword pointed to by xavailable will contain the amount of free X memory space left in the DSP and yavailable will contain the same for Y memory space. Free memory for both X and Y will always begin at physical location 0. Remember that since Program space overlays both X and Y space, the low 64 words of Y memory are used for interrupt vectors.

OPCODE 107

```
Dsp_Reserve(xreserve, yreserve)
long xreserve;
long yreserve;
```

Dsp_Reserve sets aside DSP memory for a DSP program. The amount of requested memory should not exceed the amount given by the Dsp_Available call. This function must be called to insure that your DSP process is not overwritten by a DSP subroutine which may be installed in the same area. The memory area which is set aside will be preserved until another Dsp_Reserve call is made. This will allow other processes to use the DSP program residing in this reserved space. Xreserve is the amount of X memory space that is requested and Yreserve represents the same thing in Y memory space. A 0 return value indicates that the memory was successfully reserved. A -1 indicates an error in reserving the requested memory.

Note: The Dsp_Available and Dsp_Reserve calls are only implemented to resolve memory conflicts between programs and subroutines. The calls were not meant to act as a true memory management system within the DSP. The Dsp_Available call returns the amount of memory in DSP ram available for program use that is not currently being used by subroutines. This available amount will be returned no matter how much memory is reserved by the Dsp_Reserve call. This amount will be changed if another subroutine is loaded. The Dsp_Reserve call is only used to let the system know if there is room to load another subroutine. The amount of memory reserved for programs can be changed by simply making another Dsp_Reserve call with more or less memory to reserve.

OPCODE 108

```
status = Dsp_LoadProg(file,ability,
                              buffer)
char *file;
int ability;
int status;
char *buffer
```

Dsp_LoadProg will load from disk a program to be executed in the DSP. The program must be in the ascii ".lod" format and cannot exceed the amount of space reserved by the Dsp_Reserve command. File should point to the name of the program file to be loaded into the DSP. Ability represents the 16 bit code which describes the funcionality of the given program. Buffer should point to a block of memory where the loader can place the DSP code that it generates. The size of buffer can be calculated by the formula...

3 * (#of program/data words + (3 * #blocks in the program)).
A 0 return value indicates a successful launch. A return value of -1 indicates an error occurred before the file could be executed.

OPCODE 109

```
Dsp_ExecProg(codeptr,codesize,ability)
char *codeptr;
long codesize;
int ability;
```

Dsp_ExecProg executes a DSP program which resides in binary format in memory. This function is much faster than Dsp_LoadProg since it doesn't need to read the file into memory and convert it from ascii to binary format. Codeptr should point to a block of binary dsp code. Codesize number of DSP words will be transferred from this location and downloaded into the DSP. The ability parameter specifies the programs functional ability. Codesize should not exceed the amount of memory reserved by the Dsp_Reserve call.

OPCODE 110

```
Dsp_ExecBoot(codeptr, codesize, ability)
char *codeptr;
long codesize;
int ability;
```

Dsp_ExecBoot will download into the 512 words of internal DSP memory a bootstrap program. A reset will be performed on the DSP before the program is loaded. This program can either run as a program or be used to load a larger DSP program. Note that this call currently exists for developmental test purposes only. **Only debuggers or similar programs wanting to take over the entire DSP system should use this call.** Applications should use Dsp_LoadProg and Dsp_ExecProg instead. Codeptr should point to a block of binary DSP code. Codesize number of DSP Words will be transferred from this location and downloaded into the DSP (See function Dsp_GetWordSize for a description of a DSP word). Only the first 512 DSP words of code will be downloaded.

OPCODE 111

```
size = Dsp_LodToBinary(file, codeptr)
char *file;
char *ptr;

long size
```

Dsp_LodToBinary reads in the ".lod" file whose file name is given in the variable file. The function will then convert the file into binary form ready to sent to the Dsp_ExecBoot or the Dsp_ExecProg function. Codeptr should point to a block of memory which is large enough for the routine to place the binary code data. The function will return the size of the program in DSP words. A negative size means that an error occurred during the conversion process.

OPCODE 112

```
Dsp_TriggerHC(vector);
int vector;
```

Dsp_TriggerHC will cause a host command which is set aside for DSP programs to be executed. Only two HC vectors are available to use by DSP programs. Vectors $13 and $14. All other Host vectors are used by the system and by DSP subroutines. Note that when a program is loaded for execution, the vector table is overlayed with the system's vector table. All other vectors except $13 and $14 will be overwritten by the system.

OPCODE 113

```
ability = Dsp_RequestUniqueAbility();

int ability;
```

Dsp_RequestUniqueAbility provides a way for host processes to uniquely identify their own DSP process which does not fall under a known ability definition. Upon return, the system will pass back an ability identifier which is unique to the current system session. Using this value in calls such as Dsp_InqSubrAbility will allow the host process to check to see if your code is still resident in the DSP making it unnecessary to load it back in.

OPCODE 114

```
ability = Dsp_GetProgAbility()

int ability;
```

Dsp_GetProgAbility will return to the calling process the ability of the program currently residing in the DSP. This ability value can then be used to determine if another DSP program needs to be downloaded into the DSP or if the current DSP program will do the required job.

OPCODE 115

Dsp_FlushSubroutines()

Dsp_FlushSubroutines can be called if the host process needs more DSP memory than what is returned by Dsp_Available. When this call is made, all DSP subroutines currently residing in the DSP will be removed and the memory will be returned back to the pool of usable program memory. Dsp_Available may then be called again to find out how much memory was returned to the system. Programs should make an effort to get by with the memory left in the system without making this call whenever possible. Overall system performance can be greatly enhanced if frequently called DSP code can be left in the DSP instead of having to repeatedly download them.

OPCODE 116

```
handle = Dsp_LoadSubroutine(ptr, size,
                                    ability);

char *ptr;
long size;
int ability;
```

Dsp_LoadSubroutine will install a DSP subroutine into the system to be executed at a later time. Ptr must point to a block of DSP subroutine code. This code must meet the "DSP subroutine" requirements as explained in an earlier section of this document. The size of this subroutine as well as its ability are reported in the remaining 2 variables. Dsp_LoadSubroutine will return a positive handle if the subroutine was installed without problems. A zero handle will be returned if the system was not able to install the subroutine. The subroutine will remain resident in the DSP until all of the subroutine slots have been filled and it is replaced by another subroutine. It may also be removed if a process makes a Dsp_FlushSubroutine call.

OPCODE 117

handle = Dsp_InqSubrAbility(ability);
int ability;

int handle;

Dsp_InqSubrAbility will return the handle of an installed
subroutine if the subroutine's ability matches the ability
passed into the routine. By finding a subroutine which
already exists on the DSP (whether or not the process is the
one that installed it) the calling process will save the time
taken to download it to the DSP. If the system does not find
a DSP subroutine whose ability matches the requested one, a
zero handle will be returned. In that case it would be
necessary for the calling process to use the
Dsp_LoadSubroutine call to install their own subroutine.

OPCODE 118

status = Dsp_RunSubroutine(handle);
int handle;

Dsp_RunSubroutine will execute a DSP resident subroutine
identified by the given handle. Before this call can be made
the subroutine must be identified through either a
Dsp_InqSubrAbility call or a Dsp_LoadSubroutine call. The
status which is returned from the call lets the calling process
know if the DSP subroutine was properly launched. A
negative status reports that an error occurred and that the
process was not launched. A zero return value represents a
successful launch.

OPCODE 119

hf0_ret = Dsp_Hf0(flag)
int flag;

int hf0_ret;

Dsp_Hf0 will read from or write to bit #3 of the HSR. If flag
is either a zero or a one, the value of flag will be written into
the HSR bit. If flag contains a 0xffff, the routine will return

into hf0_ret the value of bit #3 in the HSR (either 0 if cleared, 1 if set) without changing its value.

OPCODE 120

```
hf1_ret = Dsp_Hf1(flag)
int flag;

int hf1_ret;
```

Identical to Dsp_Hf0 except sets/checks bits for bit #4 of the HSR.

OPCODE 121

```
hf2_ret = Dsp_Hf2()

int hf2_ret;
```

Returns the value of bit #3 in the HCR. Note that this bit can only be read by the host and cannot be set.

OPCODE 122

```
hf3_ret = Dsp_Hf3()

int hf3_ret;
```

Similar to Dsp_Hf2 except returns value of bit #4 of the HCR.

OPCODE 125

```
status = Dsp_HStat()

char status;
```

Dsp_Hstat returns the value of the DSP's ISR port. This call enables the calling process to know whether or not the host port is ready to transmit or receive data. Please refer to the DSP56000 Users manual for a complete description of the ISR register.

Motorola DSP Assembler

Introduction

The Motorola DSP Assemblers are programs that process assembly language source statements written for Motorola's family of digital signal processors. The assembler translates these source statements into object programs compatible with other Motorola DSP software and hardware products.

Assembly Language

The assembly language provides mnemonic operation codes for all machine instructions in the digital signal processor instruction set. In addition, the assembly language contains mnemonic directives which specify auxiliary actions to be performed by the assembler. These directives are not always translated into machine language. The assembly language enables the programmer to define and use macro instructions which replace a single statement with a predefined sequence of statements found in the macro definition. Conditional assembly also is supported.

Running the Assembler

The general format of the command line to invoke the assembler is:

```
DSPASM [options] <filenames>
```

The breakdown of this command is discussed in the pages that follow.

DSPASM

The name of the Motorola DSP assembler program appropriate for the target processor.

For example, for the Motorola DSP56000 processor the name of the assembler executable is ASM56000.

[options]

Any of the following command line options. These can be in
any order, but must precede the list of source filenames.
Some options can be given more than once; the individual
descriptions indicate which options may be specified
multiple times. Option letters can be in either upper or lower
case.

Option arguments may immediately follow the option letter
or may be separated from the option letter by blanks or tabs.
However, an ambiguity arises if an option takes an optional
argument. Consider the following command line:

ASM56000 -B MAIN IO

In this example it is not clear whether the file MAIN is a
source file or is meant to be an argument to the -B option. If
the ambiguity is not resolved the assembler will assume that
MAIN is a source file and attempt to open it for reading.
This may not be what the programmer intended.

There are several ways to avoid this ambiguity. If MAIN is
supposed to be an argument to the -B option it can be placed
immediately after the option letter:

ASM56000 -BMAIN 1

If there are other options on the command line besides those
that take optional arguments the other options can be placed
between the ambiguous option and the list of source file
names:

ASM56000 -B MAIN -V 1

An alternative is to use two successive hyphens to indicate
the end of the option list:

ASM56000 -B -- MAIN 1

In this latter case the assembler interprets MAIN as a source
file name and uses the default naming conventions for the -B
option.

-A

This option indicates that the assembler should run in absolute mode, generating an absolute object file when the -B command line option is given. By default the assembler produces a relocatable object file that is subsequently processed by the Motorola DSP linker.

-B[<objfil>]

This option specifies that an object file is to be created for assembler output. <objfil> can be any legal operating system filename, including an optional pathname. A hyphen also may be used as an argument to indicate that the object file should be sent to the standard output.

The type of object file produced depends on the assembler operation mode. If the -A option is supplied on the command line, the assembler operates in absolute mode and generates an absolute object (.CLD) file. If there is no -A option on the command line, the assembler operates in relative mode and creates a relocatable object (.CLN) file.

If a pathname is not specified, the file will be created in the current directory. If no filename is specified, the assembler will use the basename (filename without extension) of the first filename encountered in the source input file list and append the appropriate file type (.CLN or .CLD) to the basename. If the -B option is not specified, then the assembler will not generate an object file. The -B option should be specified only once. If the file named in the -B option already exists, it will be overwritten. Example:

```
ASM56000 -Bfilter main.asm fft.asm
      fio.asm
```

In this example, the files MAIN.ASM, FFT.ASM, and FIO.ASM are assembled together to produce the relocatable object file FILTER.CLN.

-D<symbol> <string>

This is equivalent to a source statement of the form:

```
DEFINE <symbol> <string>
```

<string> must be preceded by a blank and should be enclosed in single quotes if it contains any embedded blanks. Note that if single quotes are used they must be passed to the assembler intact, e.g. some host command interpreters will strip single quotes from around arguments. The -D<symbol> <string> sequence can be repeated as often as desired. Example:

```
ASM56000 -D POINTS 16 prog.asm
```

All occurrences of the symbol POINTS in the program PRO.ASM will be replaced by the string '16'.

-F<argfil>

Indicates that the assembler should read Command line input from <argfil>. <argfil> can be any legal operating system filename, including an optional pathname. <argfil> is a text file containing further options, arguments, and filenames to be passed to the assembler. The arguments in the file need be separated only by some form of white space (blank, tab, newline). A semi-colon (;) on a line following white space makes the rest of the line a comment.

The -F option was introduced to circumvent the problem of limited line lengths in some host system command interpreters. It may be used as often as desired, including within the argument file itself. Example:

```
ASM56000 -Fopts.cmd
```

Invoke the assembler and take Command line options and source filenames from the command file OPTS.CMD.

-G

Send source file line number information to the object file.
This option is valid only in conjunction with the -B command
line option. The generated line number information can be
used by debuggers to provide source-level debugging.
Example:

ASM56000 -B -G myprog.asm

Assemble the file MYPROG.ASM and send source file line
number information to the resulting object file MYPROG.CLN

-I*<pathname>*

When the assembler encounters INCLUDE files, the current
directory (or the directory specified in the INCLUDE
directive) is first searched for the file. If it is not found and
the -I option is specified, the assembler prefixes the filename
(and optional pathname) specified in the INCLUDE directive
with <pathname> and searches the newly formed directory
pathname for the file.

The pathname must be a legal operating system pathname.
The -I option may be repeated as many times as desired. The
directories will be searched in the order specified on the
command line. Example:

ASM56000 -I\project\ testprog

This example uses IBM PC pathname conventions, and would
cause the assembler to prefix any INCLUDE files not found in
the current directory with the \project\ pathname.

-L*<lstfil>*

This option specifies that a listing file is to be created for
assembler output. <lstfil> can be any legal operating system
filename, including an optional pathname. A hyphen also
may be used as an argument to indicate that the listing file
should be sent to the standard output, although the listing file
is routed to standard output by default.

If a pathname is not specified, the file will be created in the
current directory. If no filename is specified, the assembler
will use the basename (filename without extension) of the
first filename encountered in the source input file list and
append .LST to the basename. If the -L option is not
specified, then the assembler will route listing output to the
standard output (usually the console or terminal screen) by
default. The -L option should be specified only once. If the
file named in the -L option already exists, it will be
overwritten. Example:

ASM56000 -L filter.asm gauss.asm

In this example, the files FILTER.ASM and GAUSS.ASM are
assembled together to produce a listing file. Because no
filename was given with the -L option, the output file will be
named using the basename of the first source file, in this case
FILTER. The listing file will be called FILTER.LST.

-M<pathname>

This is equivalent to a source statement of the form:

MACLIB <pathname>

The pathname must be a legal operating system pathname.
The -M option may be repeated as many times as desired.
The directories will be searched in the order specified on the
command line. Example:

ASM56000 -M fftlib/ trans.asm

This example uses UNIX pathname conventions, and would
cause the assembler to look in the fftlib subdirectory of the
current directory for a file with the name of the currently
invoked macro found in the source file.

-O<opt>[,<opt>,...,<opt>]

This is equivalent to a source statement of the form...

OPT <opt>[,<opt>,...,<opt>]

<opt> can be any of the options that are available with the OPT directive. If multiple options are specified, they must be separated by commas. The -O<opt> sequence can be repeated for as many options as desired.
Example:

ASM56000 -OS,CRE myprog.asm

This will activate the symbol table and cross reference listing options.

-R<rev>[,<rev>,...,<rev>]

Run the assembler without the specified processor revision level enhancements. This is for backward compatibility so that the assembler will flag new constructions as illegal. <rev> can be any of the revision specifiers given below, but must be appropriate for the target processor. If multiple revision levels are specified, they must be separated by commas. The -R<rev> sequence can be repeated for as many revision levels as desired. Example:

Processor	Revision
DSP56000	C, 2
DSP96000	B, 1

ASM56000 -RC myprog.asm

Assemble MYPROG.ASM without the DSP56000 Revision C enhancements.

-V

This option causes the assembler to report assembly progress (beginning of passes, opening and closing of input files) to the standard error output stream. This is useful to insure that assembly is proceeding normally. Example:

ASM56000 -V myprog.asm

Assemble the file MYPROG.ASM and send progress lines to the standard error output.

-Z

This option causes the assembler to strip symbol information from the absolute load file. Normally symbol information is retained in the object file for symbolic reference purposes. Note that this option is valid only when the assembler is in absolute mode via the -A command line option and when an object file is created (-B option).

Example:

ASM56000 -A -B -Z myprog.asm

Assemble the file MYPROG.ASM in absolute mode and strip symbol information from the load file created as output.

<filenames>

A list of operating system compatible filenames (including optional pathnames). If no extension is supplied for a given file, the assembler first will attempt to open the file using the filename as supplied. If that is not successful the assembler appends .ASM to the filename and attempts to open the file again. If no pathname is specified for a given file, the assembler will look for that file in the current directory. The list of files will be processed sequentially in the order given and all files will be used to generate the object file and listing.

The assembler will redirect the output listing to the standard output if the output listing is not suppressed with the IL option, or if it is not redirected via the -L command line option described above. The standard cut generally goes to the console or terminal screen by default, but can be diverted to a file or to a printer by using the I/O redirection facilities of the host operating system, if available. Error messages will always appear on the standard output, regardless of any option settings. Note that some options (-B , -L) allow a hyphen as an optional argument which indicates that the corresponding output should be sent to the standard output stream. Unpredictable results may occur if, for example, the object file is explicitly routed to standard output while the listing file is allowed to default to the same output stream.

Motorola DSP Linker

Introduction

The Motorola DSP Linker is a program that processes
relocatable object files produced by the Motorola DSP
assemblers, generating an absolute executable file which can
be loaded directly into one of the Motorola DSP simulators,
downloaded to an application development system, or
converted to Motorola S-record format for PROM burning.
A command line option provides for specification of a base
address for each DSP memory space and logical location
counter (high, low, default). In addition, a memory control
file may be supplied to indicate absolute positioning of
sections in DSP memory as well as physical mappings to
internal and external memory. The linker optionally
generates a map file which shows memory assignment of
sections by memory space and a sorted list of symbols with
their load time values.

Running the Linker

The general format of the command line to invoke the linker
is:

    DSPLNK [options] <filenames>

The breakdown of this command is discussed in the pages
that follow.

[options]

Any of the following command line options. These can be in
any order, but must precede the list of source filenames.
Some options can be given more than once; the individual
descriptions indicate which options may be specified
multiple times. Option letters can be in either upper or lower
case.

Option arguments may immediately follow the option letter
or may be separated from the option letter by blanks or tabs.

However, an ambiguity arises if an option takes an optional argument. Consider the following command line:

DSPLNK -B MAIN IO

In this example it is not clear whether the file MAIN is an input file or is meant to be an argument to the -B option. If the ambiguity is not resolved the linker will assume that MAIN is an input file and attempt to open it for reading. This may not be what the programmer intended.

There are several ways to avoid this ambiguity. If MAIN is supposed to be an argument to the -B option it can be placed immediately after the option letter:

DSPLNK -BMAIN IO

If there are other options on the command line besides those that take optional arguments the other options can be placed between the ambiguous option and the list of input file names:

DSPLNK -B MAIN -V IO

An alternative is to use two successive hyphens to indicate the end of the option list :

DSPLNK --B MAIN IO

In this case the linker interprets MAIN as an input file name and uses the default naming conventions for the -B option.

-B[<objfil>]

This option specifies that an object file is to be created for linker output.   <objfil> can be any legal operating system filename, including an optional pathname. A hyphen also may be used as an argument to indicate that the object file should be sent to the standard output.

If a pathname is not specified, the file will be created in the current directory. If no filename is specified, or if the -B option is not present, the linker will use the basename (filename without extension) of the first filename

encountered in the input file list and append .CLD to the basename. If the -l option is present (see below) an explicit filename must be given. This is because if the linker followed the default action it possibly could overwrite one of the existing input files. The -B option should be specified only once. If the file named in the -B option already exists, It will be overwritten. Example :

DSPLNK -Bfilter.cld main.cln fft.cln fio.cln

In this example, the files MAIN.CLN, FFT.CLN, and FIO.CLN are linked together to produce the absolute executable file FILTER.CLD.

-F<argfil>

Indicates that the linker should read command line input from <argfil>. <argfil> can be any legal operating system filename, including an optional pathname. <argfil> is a text file containing further options, arguments, and filenames to be passed to the linker. The arguments in the file need be separated only by some form of white space (blank, tab, newline). A semicolon (;) on a line following white space makes the rest of the line a comment.

The -F option was introduced to circumvent the problem of limited line lengths in some host system command interpreters. It may be used as often as desired, including within the argument file itself. Example:

DSPLNK -Fopts.cmd

Invoke the linker and take command line options and input filenames from the command file OPTS.CMD.

-G

Send source file line number information to the object file.
The generated line number information can be used by
debuggers to provide source-level debugging. Example:

DSPLNK -B -G myprog.cln

Link the file MYPROG.CLN and send source file line number
information to the resulting object file MYPROG.CLD.

-I

The linker ordinarily produces an absolute executable file as
output. When the -I option is given the linker combines the
input files into a single relocatable object file suitable for
reprocessing by the linker. No absolute addresses are
assigned and no errors are issued for unresolved external
references. Note that the -B option must be used when
performing incremental linking in order to give an explicit
name to the output file. If the filename were allowed to
default it could overwrite an existing input file. Example:

DSPLNK -I -B filter.cln main.cln
fft.cln fio.cln

In this example. the files MAIN.CLN, FFT.CLN, and
FIO.CLN are combined to produce the relocatable object file
FILTER.CLN .

-L<library>

The linker ordinarily processes a list of input files which each
contain a single relocatable code module. If the -L option is
encountered the linker treats the following argument as a
library file and searches the file for any outstanding
unresolved references.

If a module is found in the library that resolves an
outstanding external reference, the module is read from the
library and included in the object file output. The linker
continues to search a library until all external references are
resolved or no more references can be satisfied within the

current library. The linker searches a library only once, when
it is encountered on the command line. Therefore, the
position of the -L option on the command line is significant.
Example:

DSPLNK -B filter main fir -Lio

This example illustrates linking with a library. The files
MAIN.CLN and FIR.CLN are combined with any needed
modules in the library IO.LIB to create the file FILTER.CLD.

-M[<mapfil>]

This option indicates that a map file is to be created.
<mapfil> can be any legal operating system filename,
including an optional pathname. A hyphen also may be used
as an argument to indicate that the map file should be sent to
the standard output.

If a pathname is not specified, the file will be created in the
current directory. If no filename is specified, the linker will
use the basename (filename without extension) of the first
filename encountered in the input file list and append .MAP
to the basename. If the -M option is not specified, then the
linker will not generate a map file. The -M option should be
specified only once. If the file named in the -M option
already exists, it will be overwritten. Example:

DSPLNK -M -- filter.cln gauss.cln

In this example, the files FILTER.CLN and GAUSS.CLN are
linked together to produce a map file. Because no filename
was given with the -M option, the output file will be named
using the basename of the first input file, in this case FILTER.
The map file will be called FILTER.MAP.

-N

The linker considers case significant in symbol names. When the -N option is given the linker ignores case in symbol names; all symbols are mapped to lower case. Example :

```
DSPLNK -N filter.cln fft.cln fio.cln
```

In this example, the files FILTER.CLN, FFT.CLN, and FIO.CLN are linked to produce the absolute executable file FILTER.CLD. All symbol references are mapped to lower case.

-O<mem>[<ctr>][<map>] :<origin>

By default the linker generates instructions and data for the output file beginning at absolute location zero for all DSP memory spaces. This option allows the programmer to redefine the start address for any memory space and associated location counter.

<mem> is one of the single-character memory space identifiers (X, Y, L, P). The letter may be upper or lower case. The optional <ctr> is a letter indicating the high (H) or low (L) location counters. If no counter is specified the default counter is used. <map> is also optional and signifies the desired physical mapping for all relocatable code in the given memory space. It may be I for internal memory, E for external memory, R for ROM, A for port A, and B for port B. If <map> is not supplied, then no explicit mapping is presumed.

The <origin> is a hexadecimal number signifying the new relocation address for the given memory space. The -O option may be specified as many times as needed on the command line. This option has no effect if incremental linking is being done (see the -l option). Example:

```
DSPLNK -Ope:200 myprog -Lmylib
```

This will initialize the default P memory counter to hex 200 and map the program space to external memory.

-P<pathname>

When the linker encounters input files, the current directory
(or the directory given in the library specification) is first
searched for the file. If it is not found and the -P option is
specified, the linker prefixes the filename (and optional
pathname) of the file specification with <pathname> and
searches the newly formed directory pathname for the file.
The pathname must be a legal operating system pathname.
The -P option may be repeated as many times as desired.
The directories will be searched in the order specified on the
command line. Example :

DSPLNK -P\project\ testprog

This example uses IBM PC pathname conventions, and
would cause the linker to prefix any library tiles not found in
the current directory with the \protect\ pathname.

-R[<ctlfil>]

This option indicates that a memory control file is to be read
to determine the placement of sections in DSP memory and
other linker control functions.   <ctlfil> can be any legal
operating system filename, including an optional pathname.

If a pathname is not specified, an attempt will be made to
open the file in the current directory. If no filename is
specified, the linker will use the base name (filename without
extension) of the first filename encountered in the link input
file list and append .CTL to the basename. If the -R option is
not specified, then the linker will not use a memory control
file. The -R option should be specified only once.   Example:

DSPLNK -Rproj filter.cln gauss.cln

In this example, the files FILTER.CLN and GAUSS.CLN are
linked together using the memory file PROJ.CTL.

-U<symbol>

Allows the declaration of an unresolved reference from the command line. <symbol> must be specified. This option is useful for creating an undefined external reference in order to force linking entirely from a library. Example :

```
DSPLNK -U start -Lproj.lib
```

Declare the symbol START undefined so that it will be resolved by code within the library PROJ.LIB.

-V

This option causes the linker to report linking progress (beginning of passes, opening and closing of input files) to the standard error output stream. This is useful to insure that link editing is proceeding normally. Example :

```
DSPLNK -V myprog.cln
```

Link the file MYPROG.CLN and send progress lines to the standard error output.

-X<opt>[,<opt>,...,<opt>]

The -X option provides for link time options that alter the standard operation of the linker. The options are described below. All options may be preceded by NO to reverse their meaning. The -X<opt> sequence can be repeated for as many options as desired.

Option	Meaning
XC	Relative terms from different sections used in an expression cause an error
RSV	Reserve special target processor memory areas (e.g. DSP96000 DMA)
AEC	Check form of address expressions
RO	Allow region overlap
ESO	Do not allocate memory below ordered sections
ASC	Enable absolute section bounds checking

Example:

DSPLNK -XNOXC myprog.asm

This will disable checking of relative terms from different sections in arithmetic expressions.

-Z

The linker strips source file line number and symbol information from the output file. Symbol information normally is retained for debugging purposes. This option has no effect if incremental linking is being done (see the -l option). Example:

DSPLNK -Z filter.cln fft.cln fio.cln

In this example, the files FILTER.CLN, FFT.CLN, and FIO.CLN are linked to produce the absolute object file FILTER.CLD. The output file will contain no symbol or line number information.

<filenames>

A list of operating system compatible filenames (including optional pathnames). If no extension is supplied for a given file, the linker first will attempt to open the file using the filename as supplied. If that is not successful the linker appends .CLN to the filename and attempts to open the file again. If no pathname is specified for a given file, the linker will look for that file in the current directory. The list of files will be processed sequentially in the order given and all files will be used to generate the object file and map listing.